UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des Sciences
Département d'informatique

Efficient Verification of Counting Abstractions for Parametric Systems

Dissertation présentée en vue de l'obtention du grade de Docteur en Sciences

Année Académique 2003-2004

Laurent Van Begin

This document has been typeset under LaTeX 2_ε, using the `book` document class.

Acknowledgements

My foremost thanks go to my director Jean-François Raskin for his guidance, encouragement and ongoing support throughout this thesis. He introduced me into his research on the verification of infinite-state systems and, despite his crowded schedule, he always found some time to spend for (very interesting) discussions or to help me to overcome problems. His help was really significant to write this thesis.

I am very grateful to my co-director Thierry Massart. He initiated me to research during my master thesis, and then found grants that gave me the opportunity to work at the *Université Libre de Bruxelles*. Without him, I would have never write this manuscript.

I also thank both my director and my co-director for the enjoyable moments we spent together.

I would like to make a special thanks to Raymond Devillers who spent a lot of time by carefully, and patiently, reading this thesis. His relevant comments and ideas significantly improved my thesis.

I wish to also thank the other members of the Jury.

Giorgio Delzanno gave me the opportunity to work at the *Università di Genova* for months. I thank him for his support during my stays, the productive collaboration, and also for the not work related moments which were often very funny.

Alain Finkel invited me twice in the *École Normale Supérieure de Cachan*. I thank him for its cordial welcome and the opportunity to work on the Fast project. I learned many interesting things during my stays in Cachan.

Thanks to Pierre Ganty, Gilles Geeraerts and Alexandre Genon for fruitful discussions and the funny moments we had during my stays in Genoa. I particularly thanks Gilles Geeraerts for its collaboration on the *Concurrent Boolean Programs*.

I would like to thank the members of the *École Normale Supérieure de Cachan*, that kindly welcome me.

Thanks to all the members of the *Formal Methods and Verification group* of the *département d'Informatique* of the ULB for creating an inspiring and comfortable research atmosphere. All the members of the *département d'Informatique* of the ULB made the past four years very enjoyable.

Finally, I would like to thank my mother and my brother for their love and their encouragement.

Contents

Chapter 1

Introduction

1.1 Context

The advances in computer science and especially in hardware led to an increase of the program complexity. Nowadays, concurrency and distribution are customarily used in programming of computer systems. Programs are composed of several components running in parallel, possibly on different locations. Those concepts introduce non determinism in the behaviours of programs introduced that significantly increases the difficulties to develop *correct* programs. Non-determinism allows systems to have several possible executions.

On the other hand, correctness of programs has always been a challenging problem. A partial solution to this problem consists to use methodologies, that precisely describe all the steps leading to the construction of reliable systems. At each step, those methodologies uniformly describe a set of tasks that must be accomplished. Most of the time, a test phase, where the program is tested on a (more or less arbitrary) set of examples is used to it. These methodologies proved their utility for a large class of systems, like data bases and information systems. However, they also showed their limits when applied to the construction of concurrent and distributed systems like embedded systems, nuclear plan controllers, etc. For these systems, the management of non deterministic behaviours by designers, at a conceptual level, and by developers, at an implementation level, is still a challenging problem. Many critical systems (i.e. systems for which a failure is unacceptable because it can put human lives in danger or has severe economic consequences) fall into this class of systems. As a consequence, new techniques must be developed to construct such systems in a way that their reliability can be ensured.

Formal methods have been proposed as a way to obtain guarantees on the correctness of safety critical systems. A formal method usually proposes:

- a formal language to describe the behaviours of system, often called the model M of the system;
- a formal language to describe the properties that must be ensured by the system, often called the correctness specification φ of the system;
- a method usually systematic, to decide if the properties are verified by all the behaviours of the system, that is decide if $M \models \varphi$ holds.

A system is *correct* if it satisfies all its specified properties. Most of the time, the model is an (over)approximation of the real system which ensures that if the model satisfies the correctness specification, so does the real system. If the model does not satisfy the correctness specification, this is often connected to a mistake in the real system. Nevertheless, the models are sometimes too coarse and do not satisfy some correctness properties, although the real systems do. In those cases, the model of the system must be refined to get closer to the real systems. This is called a "*refinement step*".

We discuss next some important aspects of formal methods.

Modeling systems with mathematical models. In order to model concurrent programs, we can use transition systems.

Two main ingredients of a transition system are *states* and *transitions*. The states of the transition system model the possible configurations of the program can be and the transitions model how the configurations evolve during the executions of the program. The set of states that models the possible configurations of the system can be either a finite or an infinite set. In this thesis, we will study transition systems with an infinite set of states

Infinite state models are natural models for programs that manipulate integer variables or that can dynamically generate threads of execution. In these cases the number of configurations of the real program that will be executed is bounded due to the range of integers or the maximal number of threads of the particular platform on which it will be executed. However, we want to verify the logic of the program and hence we want to abstract it from the particular platform on which it will be executed.

Techniques to verify models. Several techniques have been proposed in the literature to verify finite and infinite state models. In both cases, the verification task is complex and difficult to be carried out by hand. As a consequence, fully automatic verification techniques encounter a great success. Automatic techniques are often based on the so-called *Model Checking* technique. Model checking consists in an exhaustive search into the state space of the model to decide if the desired properties are verified or not.

In the case of infinite state space, the model checking algorithms must manipulate infinite sets of states. To do so, those algorithms use finite symbolic representations of infinite sets of states. Those representations describe sets without enumerating their elements. Moreover, operations on the elements of sets are computed at once on all the elements without enumerating them.

For instance, suppose that the configurations are the values of two integer variables x and y. Infinite sets of such configurations can be finitely described using constraints. As an example, the constraint $x = 4 \wedge y \geq 0$ represents the set of all the valuations of x and y where x is equal to 4 and y is non-negative. Operations on elements of sets can be computed by manipulating constraints. In our example, the constraint describing the set of configurations after incrementing x by one is obtained by replacing x by $x-1$ into the constraint $x = 4 \wedge y \geq 0$. This leads to the constraint $x = 5 \wedge y \geq 0$ that describes exactly the set of configurations after the increment.

Parameterized verification. When modelling, for instance, client-server applications, i.e. applications composed of a server that answers to requests of clients, the number of clients is usually not known a priori.

To describe this kind of systems, parameterized models (also called parameterized systems) have been introduced. Those models are composed of a finite number of types of components and a number, possibly a parameter, of instances for each type. We usually want to verify such systems for any number of instances.

Verifying properties on parameterized systems corresponds to verify that those properties are ensured by any instances, i.e. for any possible values of the parameters. In the case of client-server applications, that corresponds to verify properties for any number of clients.

Parameterized systems, as we will show in this thesis, are also useful to describe systems with dynamic creations of components, for instance programs with dynamic creation of threads (processes).

As explained above, natural models for parameterized systems are infinite state systems. Indeed, their set of possible configurations is composed of the sets of configurations of all their instances.

In this thesis, we propose an approach for the automatic verification of Java programs with an (unknown and unbounded) number of threads. This approach is mainly based on parameterized verification algorithms. As a first step of the approach, abstraction techniques are used to construct parameterized models. Then, as we will show, verification algorithms allow us to automatically verify an important class of properties on these models. In the next section, we present a review of the literature on parameterized verification. Then, after a brief presentation of the Java language, we present the different works of the literature on the verification of multi-threaded Java programs. Finally, we present our contributions in the field of the parameterized verification of Java programs.

1.2 Related Works

The parameterized verification problem has been studied from the beginning of the nineties. Two main approaches for the verification of parameterized systems have been investigated. The first one consists of a reduction of the parameterized verification problem to the verification of finite state models, which has been extensively studied and for which mature technologies have been developed. The main problem of this approach is related to the construction of the finite state models. The finite state models must be close enough to the parameterized systems to allow to verify them. The second approach consists in directly verifying infinite state models. To make this approach practical, several problems must be solved:

1. A well-suited formalism must be defined to naturally model parameterized systems;

2. An effective symbolic representation of infinite sets of states must be defined;

3. As the algorithms for infinite state models are, most of the time, not guaranteed to terminate in theory, techniques must be defined to ensure termination of verification algorithms in practice.

Below we present the major works on the verification of parameterized systems described in the literature.

Reduction to finite state verification

Many works on the verification of parameterized systems presented in the literature consist in constructing a finite over-approximation of the parameterized system to verify, i.e. a finite state model such that the set of behaviours of this model contains at least a representation of all the behaviours of the parameterized system, and then verify the properties on the over-approximation. P. Wolper, V. Lovinfosse in [WL89] and R.P. Kurshan, K.L. McMillan in [KM89] independently proposed an inductive approach to prove that a finite state automaton is an over-approximation of a parameterized system S corresponding to the parallel composition of N finite state automata P. Suppose we have an automaton A that describes a possible over-approximation of the parameterized system S. The rule of Figure 1.1 to prove that S satisfies a property φ, i.e. to prove that the whole set of behaviours of S is included into the set of possible behaviours satisfying the property φ (described by an automaton A_φ). In

$$\begin{array}{ll} I_1 & L(P) \subseteq L(A) \\ I_2 & L(P||A) \subseteq L(A) \\ I_3 & L(A) \subseteq L(A_\varphi) \\ \hline & L(||_{j=1..i} P_i) \subseteq L(A_\varphi) \quad \forall i \geq 1 \end{array}$$

Figure 1.1: Induction rule

[BSV94, LHR97], techniques are presented to automatically construct candidate automata A to over-approximate the behaviours of parameterized systems.

A similar approach is proposed by Pnueli et al. in [PRZ01] where they propose a fully automatic approach to verify safety properties on parameterized systems composed of N simple and identical programs communicating through global variables.

Abstraction techniques, based on the construction of an abstraction function and a concretisation function, can be used to construct a finite state model from a parameterized system. In [ID99], Ip and Dill define an abstraction technique on parameterized systems composed of N finite state machines communicating through global variables over finite domains. More precisely, they map states of such parameterized systems into tuples where each dimension corresponds either to a location of a state machine or a global variable. Each dimension corresponding to a location can have the value 0, 1 or +1 meaning that the number of instances of finite state machines in this location is equal, respectively, to 0, 1 or more than one. The dimensions corresponding to a variable define the value of this variable. As abstract states are tuples over finite domains, the abstract models constructed using this technique are finite state and can thus be verified using finite state verification techniques. As we will see in the following, this approach is similar to the one presented in this thesis except that our abstract models retain the exact number of components that are in each local states and so are infinite state models. Our models are more precise and may allow us to prove more properties of parameterized systems.

A common point between the two approaches presented above is that they are incomplete, i.e. sometimes the methods do not allow us to know if the property is verified or not. Actually, the two methods often construct coarse over-approximations of the real systems on which verification techniques are applied. It follows that when the systems do not satisfy a property, so does the over-approximations. As the over-approximation could not satisfy the property because it is too coarse, no conclusion can be drawn on the real system when the property is proved to be unsatisfied on the finite state approximation.

The work of E.A. Emerson and V. Kahlon presented in [EK00] does not have this drawback. They studied a restricted class of parameterized systems composed of N finite automata communicating using a particular synchronization technique. Their main result is that for this specific class of parameterized systems, it suffices to prove that a finite (and small) number of instances of the parameterized system satisfy a property φ to prove that all the instances satisfy φ. This result is based on a bound k such that all the instance of the parameterized system satisfy a property φ if and only if all the instances with less or equal than k components satisfy φ.

Verification of infinite state models

The other approach investigated in the literature for the verification of parameterized systems consists in the verification of infinite state models. Two different ways have been investigated. The first way is to use powerful models (often Turing complete) to describe parameterized systems. These models are often very close to the real systems but the automatic verification methods are not guaranteed to terminate in theory. Most of the works related to this approach have been done in the so-called *regular model checking* framework [BW98, JN00, BJNT00] where the set of states are represented by finite automata and the transition relation of the models by transducers.

As termination of algorithms are not guaranteed in theory, techniques have been developed to ensure it in practice. In [JN00, DLS01, AJNd02], *acceleration techniques* are defined to ensure termination in practice. Acceleration techniques allow us to compute an infinite number of states of the parameterized systems in a finite number of steps. In [JN00, Tou01], *widening techniques* have been defined. The goal of widening is the same than acceleration techniques, i.e. to ensure the termination of algorithms in practice, but allow us to compute over-approximations of the state space instead of the exact state space of models.

Regular model checking suffers from two major problems: termination occurs only if the state space is regular, i.e. when it can be described by a finite automaton, and the topology of systems handled by these techniques are linear (rings, sequences, etc.). For these reasons, regular model checking have been extended in several ways. To verify systems with complex topologies, P.A. Abdulla et al. investigate in [AJNd02] tree automata to represent regular sets of states represented by trees. This extension allows us often to verify parameterized systems having a tree topology. To represent non-regular sets of states, D. Frisman and A. Pnueli propose to use pushdown automata in [FP01] to represent sets of states.

Finally, other representations (different from automata) have been investigated to represent the transition relation of parameterized systems and the sets of states. For instance, G. Delzanno uses Presburger logic in [Del00] to describe parameterized systems and the set of states manipulated by verification algorithms.

The other way is to model parameterized systems with restricted classes of infinite state models on which some verification problems are decidable. Decidable means here that, for those verification problems, there exist algorithms that are guaranteed to terminate in theory in all the cases and return the right answer to the verification problem. Most of the works related to this approach are based on the general results presented in [ACJT96, FS01]. In these works, most of the models used to verify parameterized systems are related to Petri nets

[Rei86]. Mainly, the approach is similar than in [ID99]. That is to say, only the number of instances of component types in each possible location is kept. The difference with the work presented in [ID99] is that, here, we retain the exact number of components in each state. The verification of parameterized systems composed of N finite automata communicating by *rendez-vous* is studied in [GS92] and Petri nets are used in [BCR01] to verify JAVA programs communicating through global boolean variables.

Extensions of Petri nets have also been used to model more precisely complex systems. For instance, Timed Petri nets are used in [AN01] to keep quantitative information about time for each component of the systems, and a model related to Coloured Petri nets is investigated in [BD02] to model systems composed of components with local variables (over infinite domains).

The extended Petri nets proposed in [EN98, EFM99, DB01] are motivated by the need to model complex communication techniques between the components of systems. In this thesis, we will go further in this direction.

1.3 Parameterized verification of multi threaded JAVA programs

The goal of this thesis is the automatic verification of properties on multi-threaded programs, and more precisely the definition of (*i*) good abstract models to describe programs and (*ii*) techniques to efficiently verify those models. Although all the results presented in this thesis can be applied to other programing languages, we focus on JAVA programs because multi-threading is directly included into the language and complex communication techniques between threads are proposed. Moreover, JAVA is a very popular programing language used to construct distributed applications, and more recently for embedded systems.

First, we briefly present the JAVA language. Then, we present the major works on the verification of multi-threaded JAVA programs presented in the literature. Most of these works consist in using finite state verification technologies to verify JAVA programs. As a consequence, the main drawback of these works is that a bound on the maximal number of instantiations of threads must be defined. Those techniques are usually only able to find errors and not to establish correctness for any number of threads.

Finally, we present our contributions. Mainly, we propose to use parameterized models to describe the behaviours of multi-threaded JAVA programs. To verify such models, we present symbolic verification algorithms that analyse extensions of Petri nets and use a special data-structure, called *Covering Sharing Tree*, to represent an infinite set of configurations.

1.3.1 The JAVA language

Modern programming languages provide complex mechanisms that allow programs to make computations in parallel (in the sense of the interleaving semantics). These features are important for programming concurrent, distributed and embedded systems. We focus in the following on the JAVA language, although the main results presented in this thesis are general and not specific to the JAVA language, and so could be adapted to other languages. We have chosen JAVA as it is a very popular programming languages for programming distributed systems (like web applications) and embedded systems (this was the main motivation of its designers).

The JAVA language is object oriented, i.e. everything is viewed as objects providing some services, except for the basic types (integer, boolean, double, etc.) for which the usual operations (increment, decrement, test, assignment, etc.) are defined.

The JAVA language is inherently multi-threaded (that is, concurrent). To be able to implement concurrency in a program, we need two important features: the ability to declare *independent pieces of code* able to run in parallel (this is provided in JAVA through the `Thread` class and its derived classes), and a set of *communication primitives* to let the threads synchronize or exchange messages (JAVA includes methods as `wait`, `notify`, `notifyAll` and `synchronized`, etc. providing communications between threads).

We briefly present in the following these important features of the language and some other related concepts like scheduling and distribution of code on several machines.

Creation of threads

A JAVA program starts with one thread executing a special `main` method. Another thread can be started by creating new `Thread` objects. A `Thread` object is usually constructed using the `Thread(Runnable target)` constructor, which expects an object implementing the `Runnable` interface as parameter. The `Runnable` interface only requests the object implementing it to define a `run` method. This method describes the behavior of the thread, i.e. the `run` method corresponds to the `main` method of the main thread. The control flow of a `Thread` object starts when the `start` method (defined in the `Thread` class) of the object is called. Figure 1.2 presents an example.

Another way to achieve the same result is through inheritance: an object is a thread if it inherits from the `Thread` class (as long as it implements the `start` method). In this case, the `run` method must be overloaded to obtain the desired behavior (By default, the `run` method of the class `Thread` does nothing).

Synchronisation between threads

JAVA gives access to several synchronization primitives one can use in the code of a thread. But before detailing these features, and in order to let the reader fully understand their intended use, we should mention an important characteristic of the multi-threading in JAVA. All the threads *share a common part of the memory*. For instance, all the statically- and heap-allocated objects can possibly be accessed by any thread holding a valid reference to it (that has been obtained through the parameters of the constructor, for instance).

We now easily understand the crucial need for *synchronization* primitives, in order to maintain the coherence of the shared objects through *mutually exclusive access* to these objects. To achieve this, JAVA associates a *lock* to each object. Whenever we want a thread to execute some critical section `I` manipulating an object `O`, we can protect the instructions with `synchronize(O){ I }`, for instance. The effect will be to *take* the lock associated to object `O` before executing the instructions `I`, and to *release* it afterwards. Of course, if the lock was already hold by another thread, then the thread attempting to synchronize is blocked and put in a wait-set associated to the object. Once the lock is relinquished, a thread arbitrarily chosen among the threads of the wait-set is elected to take the lock.

A thread can also choose to go to a special *sleep* state, in which it is inactive, up to its awakening by another thread. To let a thread sleep, we use the `wait` method of the `Object` class[1]. Again, this inactive state is associated to a given object (in this case, the object on which the `wait()` was called) *and* a given point of code (the place where the `wait` is called). To awaken one arbitrarily chosen thread among those that fell asleep on a given object, we

[1]The `Object` class is the super-class of every class in JAVA.

```
public class myThread implements Runnable {

   /* Some private data */

   public myThread(...){
      /* Constructor's code */
   }

   public void run() {
      /* The code here will be executed once the thread starts */
   }
}

public class mySystem {

   public static void main(String[] args)
   {
      Thread T1, T2 ;

      myThread M1 = new myThread(...) ;
      myThread M2 = new myThread(...) ;

      T1 = new Thread(M1) ;
      T2 = new Thread(M2) ;

      T1.start() ;
      T2.start() ;
   }
}
```

Figure 1.2: A simple sketch of concurrent program using JAVA's concurrency support.

```
public class O {
   public void m1(void)
   {
      /* ... */
      wait() ; /* equivalent to this.wait() ; */
   }

   public void m2(void)
   {
      /* ... */
      notify() ; /* equivalent to this.notify() ; */
   }
}
```

Figure 1.3: A simple `wait` - `notify` example.

use the `notify()` method on this object. To awaken all the threads, we use the `notifyAll()` method.

For instance, let us look at the piece of code of figure 1.3. If a thread calls method `m1` on a given instance `i` of class `O`, it will eventually fall asleep. One way to awaken it is that another thread calls the `m2` method on the same instance `i`. If this second thread calls `m2` on another instance, the `notify()` statement will have no effect on both threads.

Scheduling of Threads

A processor can execute only one thread at the same time. It follows that, when there are several threads that run together, only one of them can execute exactly at the same time. To simulate the parallel execution of all the threads, scheduling techniques are used. Mainly, cpu time is cut in several parts allocated to several threads. A piece of time can be allocated to the different threads of the programs following several ways, corresponding to different scheduling policies.

Following the semantics of JAVA, the implementations must only ensure weak properties on the scheduling policy. More precisely, threads can have different priorities and it must be only ensured that *threads having the highest priority will be eventually executed.*

A JAVA program must be *correct* for any scheduling strategy satisfying this property. As threads are not guaranteed to obtain the processor at a specific time and we focus only on safety properties (properties evaluated on finite (but unbounded) executions), we conclude that the scheduling information has no impact and can be hidden when we verify JAVA programs. For this reason, we will not talk anymore about the scheduling in the following.

Distribution of JAVA Programs

JAVA provides a very powerful tool, called RMI, for distributed applications programming. RMI is an object-oriented version of the *remote procedure call* technique. This technique allows us to invoke some methods of objects from another host in the same way as a usual method call. Actually, some subtle differences occur when using RMI method calls : static variables are duplicated (one instance on each host), some parameters of methods are passed by value, etc.

In [SL01], a translation from distributed JAVA programs using RMI to multi threaded JAVA programs with a unique process is defined. As this translation preserves reachability properties, we will assume in the following that such a translation is first applied when dealing with distributed JAVA programs.

This quick introduction should be sufficient to help the reader understand the sequel. That is why we do not look any further into the details of the implementation of concurrency in JAVA. More information can be found in [Gra97b, Fla97], for instance.

1.3.2 Verification of JAVA programs

Recently, verification of software became a very popular topic of research. As JAVA is one of the most popular programming languages for the construction of distributed systems, researchers decided to focus on the problem of the automatic verification of multi-threaded JAVA programs.

The BANDERA project [HD01] is one of the most popular projects in the topic of the verification of multi-threaded JAVA programs. It uses abstraction and reduction methods based on partial order techniques to allow the verification of properties, belonging to a linear time temporal logic, on JAVA programs of medium-large size. These abstraction techniques allow us to go from data's with infinite domain to data's with finite domain. The set of reachable states of the (abstract) program is enumeratively constructed starting from the initial state of the abstract program.

The BANDERA tool is designed to verify multi-threaded JAVA programs with a bounded number of instantiations for each class and so, a bounded number of threads. As a consequence, a bound is fixed on the number of instantiations for each class of the program. When the abstract program tries to exceed the maximum number of instantiations of a class C, it blocks. This guarantees that the analysed models are always finite.

The following projects have the same approach as BANDERA, i.e., they are designed for the verification of JAVA programs with a bounded number of threads, they use some abstraction methods and make an enumerative exploration of the state space.

- The JCAT project [DIS98] tries to find an execution leading to a deadlock, i.e. a state where threads are blocked forever;
- The YAV project [Ios00] tries to find an execution that does not satisfy some linear time temporal logic formula;
- The JAVA PATHFINDER project [VHBP00] tries to find an execution leading to a deadlock or a state of the program that does not satisfy an invariant assertion.

Other works based on an enumerative forward exploration of a finite state space are presented in [Che00], where a translation from a restricted class of JAVA programs to a process algebra is defined, and in [Sto02] and [PSSD00].

To manage the case where it is not possible to bound the number of threads running at the same time during the execution of the programs, parameterized techniques are used by T. Ball, S.K. Rajamani et al. in [BCR01]. They define a model composed of an undefined number of finite state automata communicating through a finite number of global boolean variables, they model a memory manager serving several clients (processes needing memory) and apply automatic techniques implemented into the BEACON tool to answer questions of the form *is it possible to reach bad configurations from the initial ones?* (see [BCR01] for more details). However, this work fails both in the definition of the link (translation) between (from) multi-threaded JAVA programs and (to) the abstract models and the availability to verify JAVA programs using synchronisation primitives.

The work presented in this thesis is related to the work of [BCR01] in the sense that the parameterized models used in this thesis are extensions of those ones used in [BCR01]. However, the approach for the verification of the parameterized models is different. As we will see, symbolic techniques are developed in this thesis whereas the verification algorithms presented in [BCR01] is enumerative.

1.4 Contributions

In this thesis we focus on techniques that allow us to verify properties on concurrent programs where the number of threads is unknown (and so possibly unbounded). Our techniques are based on the Petri net theory. More precisely, we propose to use abstraction techniques to construct (extended) Petri nets on which verification algorithms can be applied. We focus on verification problems on those models that can be fully decided with algorithms. This means that the verification algorithms applied on the (extended) Petri nets have theoretical termination guarantees. However, as the models that we analyse are abstractions of JAVA programs, in some cases we cannot draw direct conclusions about the program correctness: an additional analysis is necessary. We guess that in practice this case does not occur often.

More precisely, our original contributions are the following.

1. We propose a framework for the fully automatic verification of JAVA programs with an unknown (and unbounded) number of threads. Several tools have still been developed for the automatic verification of JAVA programs, see e.g. [HD01]. However, all these tools consider that the maximum number of threads running in parallel is fixed. Actually, they always use finite state model checking techniques to analyse programs.

   Our aim is the verification of programs manipulating datas over infinite domains and composed of an unknown number of threads. To achieve this goal, we present several abstraction steps leading to the construction of an infinite state abstract model where we can consider an unbounded number of threads running in parallel. The first abstraction allows us to construct programs manipulating variables overs finite domains. The second abstraction allows us to construct abstract models that allow us to describe the behaviors of programs composed of any number of threads running in parallel. An important feature of these abstract models is that they can be automatically verified.

2. We present a new extension of Petri nets, called *Multi Transfer Nets* (MTNs), well suited to model multi-threaded programs. Petri nets are intensively studied models for several reasons. First, they allow us to describe the behaviours of interesting classes of systems with an unbounded number of resources, e.g. an unbounded number of threads. Second, although Petri nets are infinite state models, a large number of properties can be automatically verified on them.

However, the expressiveness of basic Petri nets is too limited to describe precisely the behaviour of complex systems like multi threaded programs. Several extensions have been proposed (see [Cia94] for instance) to increase their expressive power. Those extensions are still not sufficient for our purpose. This is the reason why we introduce in this work MTNs, a new extension of Petri nets adequate for the modeling of multi-threaded programs. Unfortunately, we will show that those extended Petri nets are more difficult to analyse than usual Petri nets: some properties decidable on Petri nets become undecidable on MTNs.

3. We define a formal language, called *Concurrent Boolean Program*, for the description of multi-threaded programs and show how to translate such programs into MTN models that are amenable to verification. We present concurrent boolean programs as an adequate model for the description of abstractions of concurrent programs. The main characteristics of this language is that it has a formal syntax and a formal semantics; only boolean variables are allowed and an unbounded number of threads can be instantiated.

4. We present a data structure called *Covering Sharing Tree* allowing us to represent a class of infinite sets of states of systems called *closed sets*. The covering sharing trees are data structures that can be viewed as a graph structure similar to Binary Decision Diagrams (BDD) [Bry86] and *Sharing Trees* [ZC95]. This data structure is abstract in the sense that the domain of variables is not defined. Several data structures can be constructed by instantiating them. As examples, *upward closed covering sharing tree* and *downward closed covering sharing tree* are instances of covering sharing trees allowing to efficiently represent (infinite) *upward closed* and *downward closed* sets of tuples on natural numbers. These instances are particularly interesting to design efficient algorithms to verify properties on (extended) Petri nets including properties of the form "*Can we reach a set of states* Bad *from a set of initial states* S_0*?*".

   We also study the complexity of problems defined on covering sharing trees. These results are in general not particularly good. This can be viewed as a direct consequence of the compactness property of covering sharing trees. To avoid this problem in practice, we present heuristics that appear to work well.

5. We define symbolic model checking algorithms using a specialisation of the covering sharing tree data structure to represent infinite sets of tuples on natural numbers. More precisely, a forward algorithm and a backward algorithm for the verification of MTNs using respectively *downward closed covering sharing trees* and *upward closed covering sharing trees* are defined. The forward algorithm is based on the enumerative algorithm presented in [Fin93] and the backward algorithm is based on the generic algorithm presented in [ACJT96]. As the algorithm in [Fin93] cannot be extended to the full class of MTNs, we present an algorithm working only on a subclass. This sub-class includes the abstract models of Java programs that we have considered in our experiments.

   We also present an heuristics that guides the backward verification algorithm. This heuristics is based on *structural invariants* and it is used to prune the backward search. We show that structural invariants can be computed for our multi transfer nets.

   This heuristics turns out to be particularly efficient to verify covering properties on multi transfer nets.

Finally, as a practical contribution, these algorithm have been implemented into two prototypes, one for the forward approach and one for the backward approach, and evaluated. The two prototypes have been written in PASCAL and the source code of each of them has more or less 3500 lines.

A library in C for the covering sharing trees have also been implemented as a part of the BABYLON project. More details on this project can be found on the web page http://www.ulb.ac.be/di/ssd/lvbegin/CST/index.html.

1.5 Plan of the thesis

This thesis is organized as follows.

Chapter 2 *Models to Describe Parameterized Concurrent Systems.* We first present an automata-based model, called *Global Machines*, that we will use to describe the behaviours of multi-threaded programs. We also present a *lower level* model based on Petri nets, called *Multi Transfer Net*, that allows us to describe in a more abstract way the behaviours of global machines. Then, we present a study of a sub class of the multi transfer nets by exhibiting decidability and undecidability results on these multi transfer nets for a set of well-known problem on (extended) Petri nets. This subclass allows us to model multi threaded programs where threads communicate through *signals*. Signals are sent into the systems by a thread and caught by another one. The difference with the classical *rendez-vous* technique is that the sending is non-blocking, i.e. if no thread needs to catch a signal, it is lost and the sender continues its execution. Finally, we show the link between the global machines and multi transfers nets. This is achieved through the definition of the *counting abstraction* on global machines.

Chapter 3 *Abstraction of Concurrent Programs.* We present a third abstract model, called *Concurrent Boolean Program* that contains explicitly concepts close to JAVA constructs. This model allows us to describe the behaviour of multi-threaded programs. We show how to construct a concurrent boolean program from a multi threaded program by using the *predicate abstraction*. This approach is presented by exhibiting examples of abstraction of several JAVA programs taken from the literature.

Finally, we define a translation from concurrent boolean programs to global machines. This last translation allows us to have a fully automatic abstraction phase from multi threaded JAVA programs to multi transfer nets. This is particularly useful to have fully automatic techniques for the verification of multi threaded programs.

Chapter 4 *Handling Closed Sets.* We present a data structure called *Covering Sharing Tree* that allows us to finitely and efficiently represent an interesting class of infinite sets of tuples called closed sets. We start the chapter by defining logics that allows us to define *upward closed sets* and *downward closed sets* of tuples of integer values. As formula of such logics can become very large when we manipulate them in symbolic model checking algorithms, we then present symbolic representations for upward closed and downward closed sets. First, we show that data structures studied in the literature to represent infinite sets of tuples over natural numbers are not well-suited to symbolically represent upward and downward closed sets, .i.e. the size of these data-structures can be very huge when they are used to represent such sets.

Then, we present our covering sharing tree and study the complexity of operations that allows us to manipulate sets. Finally, as some theoretical results on complexity are not so positive, we finish the chapter by presenting heuristics based on simulation computation that allows us to efficiently approximate some operations.

Chapter 5 *Symbolic Verification Algorithms.* We present several algorithms to verify properties on multi transfer nets. We distinguish two kinds of algorithms. First, we present *backward algorithms* that start from a set of Bad states and compute the set of its predecessors. These algorithms manipulate upward closed set of states and are guaranteed to terminate. The other algorithms are *forward algorithms* that start from the initial states of the nets and compute (an over-approximation of) the set of their successors. These algorithms can be viewed as manipulating downward closed sets of states. The main difference with backward algorithms is that forward algorithms sometimes return a *don't know* answer when they are applied on extensions of Petri nets. That means that the algorithms sometimes cannot answer if a given property is verified or not on a multi transfer net.

In the two cases, we present symbolic algorithms that use the covering sharing trees. As backward approach seems less efficient, an heuristics is presented to guide the backward algorithms in their computations. Finally, we show how to use a combination of forward and backward approaches to speed up the verification process.

We finish the chapter by making an evaluation of the algorithms presented in the chapter.

Chapter 6 *Conclusions.* This chapter concludes the thesis and presents some future works.

Some chapters of this thesis are based on previously published material which have been revisited according to new insights by the author. The chapters originate from the following papers.

Chapter 2

- G. Delzanno, J-F. Raskin, and L. Van Begin. Towards the Automated Verification of Multithreaded Java Programs. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *LNCS*, pages 173–187. Springer, 2002.

- J.-F. Raskin and L. Van Begin. Petri Nets with Non-blocking Arcs are Difficult to Analyse. In *Proceedings of the 5th international workshop on verification of infinite-state systems (INFINITY 2003)*, 2003. Will also appear in Volume 96.1 of ENTCS. Elsevier, 2003.

Chapter 3

- G. Delzanno, J-F. Raskin, and L. Van Begin. Towards the Automated Verification of Multithreaded Java Programs. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *LNCS*, pages 173–187. Springer, 2002.

Chapter 4

- G. Delzanno, J-F. Raskin, and L. Van Begin. Covering Sharing Trees: Efficient Data Structures for the Automated Verification of Parametrized Systems. *accepted for publication in Software Tools for Technology Transfer Manuscript*, 2002.

Chapter 5

- G. Delzanno, J.-F. Raskin, and L. Van Begin. Attacking Symbolic State Explosion. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *LNCS*, pages 298–310. Springer, 2001.

- G. Delzanno, J-F. Raskin, and L. Van Begin. Towards the Automated Verification of Multithreaded Java Programs. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *LNCS*, pages 173–187. Springer, 2002.

- A. Finkel, J.-F. Raskin, M. Samuelides, and L. Van Begin. Monotonic Extentions of Petri Nets : Forward and Backward Search Revisited. In *Proceedings of the 4th international workshop on verification of infinite-state systems (INFINITY 2002)*, volume 68 of ENTCS. Elsevier, 2002.

Chapter 2

Models to Describe Parameterized Concurrent Systems

We present in this chapter the models that we consider in this thesis. As a high level model, we introduce *global machines* that are communicating finite state machines that use several techniques to communicate together. More precisely, global machines can be viewed as an extension of several models presented in the literature like finite state machines communicating through rendez-vous of [GS92], the local/global finite state machines of [BCR01] and broadcast protocols of [EN98]. Global machines can be used to model systems composed of components communicating together. Hence, any system that can be modeled with global machines could be verified by using the verification techniques presented in Chapter 5. One of the class of such systems is the multi-threaded JAVA programs. In the next chapter, we show how to model multi-threaded JAVA programs by using global machines.

We also introduce a new model called *multi-transfer nets* (MTN). That model extends Petri nets and is used as counting abstraction of global machines, i.e. to abstract global machines by only keeping the number of instances of local machines that are in each possible local state.

The two models allow us to describe special synchronization techniques usually used in multi threaded software, for instance, the behavior of the notify and the notifyAll primitives of JAVA. This aspect will be developed in Chapter 3.

The structure of the chapter is the following. We introduce global machines and MTN, then we study the decidability of some problems on these models. We finish the chapter by defining a function that allows us to abstract global machines into multi-transfer nets.

The content of this chapter is based on the papers [DRVB02, RVB03].

2.1 Global Machines

We present in this section sets of state machines, called global machines, communicating using (boolean) global variables and several synchronization techniques. A state machine, called here local machine, is composed of local states and transitions between their local states.

Global machines can be viewed as a description of an entire system composed by several components running in parallel and communicating together. A local machine describes one component of the system.

We start the formal description of global machines with the operations needed to handle global boolean variables used by local machines to communicate.

Definition 1 (Boolean guard/Boolean action) Let $\mathbf{B} = \{b_1, \ldots, b_n\}$ be a finite set of *global boolean variables*, and let $\mathbf{B}'$ be their primed versions. A *boolean guard* φ_g is either the formula *true* or the conjunction of literals $L_1 \wedge \ldots \wedge L_p$, $p \leq n$, such that L_i is either b or $\neg b$ for some $b \in \mathbf{B}$. A *boolean action* φ_a is a formula $b'_1 = v_1 \wedge \ldots \wedge b'_n = v_n$, where $v_i \in \{true, false, b_i\}$ for $i : 1, \ldots, n$.

In the following, boolean guards and boolean actions will be associated to transitions of state machines to respectively test the value of boolean variables and modify the value of boolean variables. Intuitively, state machines move from states to other ones by firing transitions labelled by boolean guards and boolean actions. A transitions can be applied only if its boolean guard is satisfied by the current valuation of variables and boolean action describes the assignments of variables applied when the transition is fired.

Boolean actions allow us to only assign *true* or *false* to boolean variables, or leave their value unchanged. However, more complicated assignments can be done. For instance, to assign the value of a variable to another one when applying a transition, we proceed as follows. Suppose that we want to assign the value of the variable b_2 to b_1. For each possible value of b_2, we will have a transition. More precisely, we define the actions $b'_1 = true$ and $b'_1 = false$. The boolean guard b_2 is associated to the first action to ensure that this action will be never applied when b_2 is equal to *false*. In the same way, the guard $\neg b_2$ is associated to the second action. Finally, instead of having one transition that assigns the value of b_2 to b_1, we will have two different transitions: one will be applied when b_2 is equal to *true* and will assign *true* to b_1, and the second one will be applied when b_2 is equal to *false* and will assign *false* to b_1.

Synchronization techniques are the other possibility of local machines to communicate together. A synchronization technique is a technique allowing several local machines to fire a transition together.

Synchronization labels are defined as follows.

Definition 2 (Synchronization label) Given a set Δ of symbols [1], we define the set of synchronization labels Σ built from Δ as the set $\Sigma = \Delta \cup \{\ell!, \ell? | \ell \in \Delta\} \cup \{\ell!!, \ell?? | \ell \in \Delta\} \cup \{\ell \uparrow, \ell \downarrow | \ell \in \Delta\}$.

The actions built from the set of boolean variables $\mathbf{B}$ and the set of synchronization labels Σ are as follows (in the following $\varphi = \varphi_g \wedge \varphi_a$ is the conjunction of a boolean guard φ_g with an action φ_a over $\mathbf{B}$, and $\ell \in \Delta$):

- *Internal action:* $\ell : \varphi$;
- *Rendez-vous:* $\ell! : \varphi$ (sending), and $\ell?$ (reception);
- *Signal:* $\ell\uparrow: \varphi$ (sending), and $\ell\downarrow$ (reception);
- *Broadcast:* $\ell!! : \varphi$ (sending), and $\ell??$ (reception).

Those actions will label the transitions of local machines. Intuitively, a transition of a local machine labeled by the action $\ell : \varphi$ means that the transition can be fired by the local machine

[1] We consider that there is no symbol of the form $\ell!, \ell?, \ell!!, \ell??, \ell \uparrow$ and $\ell \downarrow$ in Δ.

only if the current values of global boolean variables satisfy φ_g. Moreover, the values of the boolean variables are modified following the action φ_a and no synchronization occurs when the transition is fired. If the synchronization label is of the form $\ell!$, $\ell?$, $\ell\uparrow: \varphi$, $\ell\downarrow$, $\ell!! : \varphi$ or $\ell??$, then a synchronization between several local machines occurs when the transition is fired.

The finite state machines that we consider in the following are defined as follows.

Definition 3 (Local Machine) A *local machine* is a tuple $\langle Q, \Sigma, \delta\rangle$, where: Q is a finite set of states; Σ is the set of synchronization labels used to build the set of possible actions $\mathcal{A}$; and $\delta \subseteq (Q \times \mathcal{A} \times Q)$ is the local transition relation. In the following, we will use $s \xrightarrow{\alpha} s'$ to indicate that $\langle s, \alpha, s'\rangle \in \delta$. Moreover, δ satisfies the following properties:

1. For each synchronization label $a?? \in \Sigma$ we have $\{s \mid \exists s' \text{ such that } \langle s, a??, s'\rangle \in \delta\} \cap \{s \mid \exists s' \text{ such that } \langle s', a??, s\rangle \in \delta\} = \emptyset$, and for each synchronization label $a\downarrow \in \Sigma$ we have $\{s \mid \exists s' \text{ such that } \langle s, a\downarrow, s'\rangle \in \delta\} \cap \{s \mid \exists s' \text{ such that } \langle s', a\downarrow, s\rangle \in \delta\} = \emptyset$;

2. for all the actions of the form $\ell\downarrow$ or $\ell??$ and for any local states s, s' and s'', if we have $\langle s, \ell\downarrow, s'\rangle \in \delta$, respectively $\langle s, \ell??, s'\rangle \in \delta$, then we have $\langle s, \ell\downarrow, s''\rangle \notin \delta$, respectively $\langle s, \ell??, s''\rangle \notin \delta$.

Intuitively, the labels of the form $\ell??$ and $\ell\downarrow$ correspond to the reception of a message sent by another local machine. The first constraint avoids cyclic behaviours when the local machines receive several times a same signal or broadcast message (notice that, as a particular case, this constraint avoids the transitions of the form $\langle s, a??, s\rangle$ and $\langle s, a\downarrow, s\rangle$), The second constraint means that the local machines react deterministically when receiving signal or broadcast messages, i.e. a local machine in a local state s that receives a signal or a broadcast message a can move to only *one* possible local state.

In our case, those constraints will be always satisfied by the local machines constructed from JAVA programs.

Figure 2.1 shows an example of local machine. To ease the reading, the boolean guards are forgotten when equal to *true* and assignments are forgotten when they are of the form $x' = x$. This local machine has ten states represented by circles. An arrow from a state s to a state s' labeled by α means that we have $s \xrightarrow{\alpha} s'$ in the transition relation. For instance, we have $\langle \texttt{while}, incXY, \texttt{incX()}\rangle$ in the transition relation of the local machine.

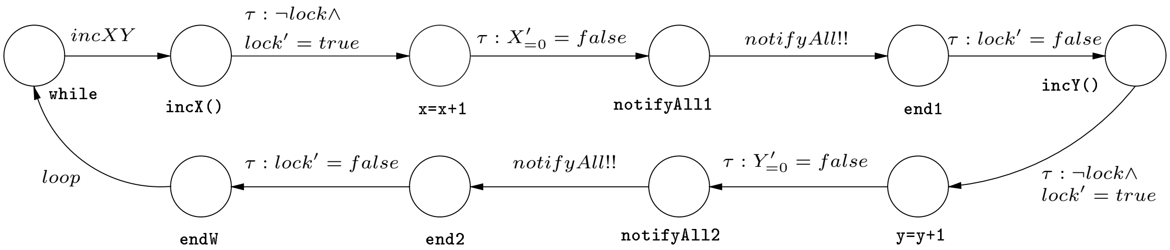

Figure 2.1: An example of Local Machine with the set of boolean variables $\{X_{=0}, Y_{=0}, lock\}$ and $\Delta = \{\tau, incXY, loop, notifyAll\}$.

Definition 4 (Global Machine) A *global machine* (GM) is a tuple $\mathcal{G} = \langle \mathbf{B}, \langle \mathcal{L}_1, k_1\rangle, \ldots, \langle \mathcal{L}_m, k_m\rangle\rangle$, where: $\mathbf{B} = \{b_1, \ldots, b_n\}$ is the set of *global boolean variables* and for $i : 1, \ldots, m$

$\mathcal{L}_i = \langle Q_i, \Sigma_i, \delta_i \rangle$ is the i-th local machine and the value k_i is the number of its copies. Furthermore, we have that $Q_i \cap Q_j = \emptyset$ for any $i, j : 1, \ldots, m$ with $i \neq j$.

Notice that if $\mathcal{L}_1$ and $\mathcal{L}_2$ are *equivalent* (i.e. they are isomorphic), then the global machine $\langle \mathbf{B}, \langle \mathcal{L}_1, k_1 \rangle, \langle \mathcal{L}_2, k_2 \rangle \rangle$ is equivalent to the global machine $\langle \mathbf{B}, \langle \mathcal{L}_1, k_1 + k_2 \rangle \rangle$.

In the following, we will always consider that Q_i, Σ_i and δ_i are respectively the set of local states, the set of synchronization labels and the transition relation of the local machine $\mathcal{L}_i$.

Definition 5 (Global State) A *global state* of $\mathcal{G} = \langle \mathbf{B}, \langle \mathcal{L}_1, k_1 \rangle, \ldots, \langle \mathcal{L}_m, k_m \rangle \rangle$ is a tuple $G = \langle \rho, \mathbf{s} \rangle$ where $\rho : \mathbf{B} \to \{true, false\}$ is a valuation for the global boolean variables, and the tuple $\mathbf{s} = \langle s_{11}, \ldots, s_{1k_1}, \ldots, s_{m1}, \ldots, s_{mk_m} \rangle$ of dimension $k = k_1 + \ldots + k_m$ is such that $s_{ij} \in Q_i$ denotes the current local state of the j-th copy of the local machine $\mathcal{L}_i$.

The runs of a global machine are defined via the relation $\Longrightarrow$ defined below.

Definition 6 (The $\Longrightarrow$ Relation) Let $\mathcal{G} = \langle \mathbf{B}, \langle \mathcal{L}_1, k_1 \rangle, \ldots, \langle \mathcal{L}_m, k_m \rangle \rangle$ be a global machine, $G = \langle \rho, \langle s_1 \ldots, s_k \rangle \rangle$ and $G' = \langle \rho', \langle s'_1 \ldots, s'_k \rangle \rangle$ be two global states, ℓ be a synchronization label and $\gamma = \rho \cup \rho'$[2]. Then, $G \stackrel{\ell}{\Longrightarrow}_{\mathcal{G}} G'$ iff one of the following conditions holds:

- there exists i such that we have $s_i \xrightarrow{\ell:\varphi} s'_i$ in the definition of one of the local machines with $\gamma(\varphi) = true$, and $s'_j = s_j$ for all $j \neq i$;
- there exist $i \neq j$ such that we have $s_i \xrightarrow{\ell!:\varphi} s'_i$ and $s_j \xrightarrow{\ell?} s'_j$ in the definitions of the local machines with $\gamma(\varphi) = true$, and $s'_r = s_r$ for all $r \neq i, j$;
- there exists i such that we have $s_i \xrightarrow{\ell\uparrow:\varphi} s'_i$ in the definition of one of the local machines, $\gamma(\varphi) = true$ and:
  - either there exists $j \neq i$ such that we have $s_j \xrightarrow{\ell\downarrow} s'_j$ in the definition of one of the local machines and $s'_r = s_r$ for any $r \neq i, j$,
  - or there are no $j \neq i$ such that we have $s_j \xrightarrow{\ell\downarrow} s'_j$ in the definition of one of the local machine, and $s'_r = s_r$ for any $r \neq i$;
- there exists i such that $s_i \xrightarrow{\ell!!:\varphi} s'_i$ in the definition of one of the local machines with $\gamma(\varphi) = true$, and for all $j \neq i$ such that there exists v with $s_j \xrightarrow{\ell??} v$ in the definition of one of the local machines we have $s'_j = v$; finally, $s'_r = s_r$ for all $r \neq i$ such that there is no v such that $s_r \xrightarrow{\ell??} v$ in the definition of one of the local machines.

Definition 7 (Run of global machine) Given a GM $\mathcal{G} = \langle \mathbf{B}, \langle \mathcal{L}_1, k_1 \rangle, \ldots, \langle \mathcal{L}_m, k_m \rangle \rangle$, a *run of the global machine* $\mathcal{G}$ is a sequence $G_1 G_2 \ldots G_n$ where G_i ($1 \leq i \leq n$) is a global state of $\mathcal{G}$ and $G_i \stackrel{\ell}{\Longrightarrow}_{\mathcal{G}} G_{i+1}$ for all $1 \leq i \leq n-1$. G_1 is called the initial global state of the run and G_n is called the target global state of the run.

[2] $\rho \cup \rho'$ defines the valuation on the variables and their primed version such that $\rho \cup \rho'(x) = \rho(x)$ for all variable x, and $\rho \cup \rho'(x') = \rho'(x')$ for all primed variable x'. $\rho \cup \rho'$ is extended to boolean guards and boolean actions in the obvious way.

A global state G' is reachable from G in $\mathcal{G}$, written $G \stackrel{*}{\Rightarrow}_{\mathcal{G}} G'$, if and only if there exists a run with initial global state G and target global state G'. The set of reachable global states from a set of initial global states $\mathbf{G}$, noted $\mathsf{Reach}_{\mathcal{G}}(\mathbf{G})$, is equal to $\{G' \mid \exists G \in \mathbf{G} : G \stackrel{*}{\Rightarrow}_{\mathcal{G}} G'\}$.

In the following, we will focus on the verification of properties on GMs of the form "*is it possible to reach a set of bad global states from an initial global state?*". In many cases, the bad states correspond to a valuation of the boolean variables satisfying a formula and a configuration of the local machines where there are *at least* a minimum number of local machines in particular local states.

Those properties are called *local state reachability properties* and the problem that consists to verify such properties is defined as follows.

Definition 8 (Local state reachability problem for global machines) The *local state reachability problem* for global machines is defined as follows.
Instance: A global machine $\mathcal{G} = \langle \mathbf{B}, \langle \mathcal{L}_1, k_1 \rangle, \ldots, \langle \mathcal{L}_m, k_m \rangle \rangle$ with the initial global state G_0, a formula φ over $\mathbf{B}$ of the propositional logic and a function $f_S : Q_1 \cup \ldots \cup Q_m \to \mathbb{N}$.
Question: Is there a global state $\langle \rho, \mathbf{s} \rangle$ reachable from G_0 such that $\rho(\varphi)$ and for all $s \in Q_1 \cup \ldots \cup Q_m$ there are at least $f_S(s)$ different instances of local machines that are in the local state s in $\mathbf{s}$?

If the answer of the local state reachability problem on a GM $\mathcal{G}$ with the initial global state G_0 for the boolean formula φ and the function f_S is *yes*, then we say that $\mathcal{G}$ *initialised in* G_0 *satisfies the local state reachability property for* φ *and* f_S. Otherwise, we say that $\mathcal{G}$ does not satisfy it.

This class of problems includes, for instance, mutual exclusion problems that consist to decide if it is possible or not to reach a global state where there is *at least* two local machines in two mutually exclusive local states. In those particular cases, φ is equal to *true* (no constraint on the boolean variables) and f_S is equal to 0 for all the local states except for the local states that are mutually exclusive. For the two mutually exclusive states, f_S is equal to 1 (at least one local machine in each state).

2.2 Multi Transfer Nets

Following [GS92, Del00, BCR01], in order to study safety properties of global machines, we will apply a *counting* abstraction that maps global states into markings that keep track of the number of processes in each one of the local states. To be able to model the communication mechanisms of Definition 3, we need an extended Petri Net model that we call Multi-transfer Nets (MTNs). MTNs have all the features of Petri Nets. In addition, MTNs allow us to capture the semantics of signals and of broadcasts as an instance of a general notion of *transfer of at most* k *tokens*.

In the rest of this thesis, we will use $\mathbb{Z}$ to denote integer numbers, $\mathbb{N}$ to denote non-negative integer numbers, and D^k to denote sets of tuples with k components taken from D. Furthermore, we will use $\mathbf{t}$ to denote a tuple $\langle t_1, \ldots, t_k \rangle$ in D^k, i.e., such that $t_i \in D$ for $1 \leq i \leq k$.

2.2.1 Definition

Formally, multi-transfer nets are defined as follows.

Definition 9 (Multi-Transfer Nets) A *Multi-Transfer Net* $\mathcal{M}$ is a pair $\langle \mathcal{P}, \mathcal{B} \rangle$ where: $\mathcal{P} = \{p_1, \ldots, p_n\}$ is a finite set of places, and $\mathcal{B} = \{M_1, \ldots, M_m\}$ is a finite set of *multi-transfers*. A *multi-transfer* M is a tuple $\langle T, \{S_1, \ldots, S_u\} \rangle$ such that

- $T = \langle \mathcal{I}, \mathcal{O} \rangle$ is the Petri Net *transition* of the *multi-transfer*, that is $\mathcal{I}, \mathcal{O} : \mathcal{P} \to \mathbb{N}$;
- each $S_k = \langle \{B_{k1}, \ldots, B_{kr_k}\}, c_k \rangle$ with $r_k > 0$ is a *transfer block*, where $c_k \in \mathbb{N} \cup \{+\infty\} \setminus \{0\}$ is the *bound*, and each $B_{kj} = \langle P_{kj}, p_{kj} \rangle$ with $P_{kj} \subseteq \mathcal{P}$ and $p_{kj} \in \mathcal{P}$ is a *transfer*. The places in P_{kj} are called the *source places* of the transfer and p_{kj} is called the *target place* of the transfer.

In order to avoid cyclic transfers, a multi-transfer M with the set of transfer blocks $\{S_1, \ldots, S_u\}$ must satisfy the following conditions:

1. for any *transfer block* S_k with the set of *transfers* $\{B_{k1}, \ldots, B_{kr_k}\}$, and for any such B_{kj}, we require that $p_{kj} \notin P_{kj}$;
2. for any *transfer* B_{ki} in the transfer block S_k and B_{lj} in the transfer block S_l with $B_{ki} \neq B_{lj}$, we require that $(P_{ki} \cup \{p_{ki}\}) \cap (P_{lj} \cup \{p_{lj}\}) = \emptyset$.

Figure 2.2 shows an example of MTN. Places are represented by circles, i.e. the example is composed of the set of places $\{p_1, p_2, p_3, p_4\}$, and multi-transfers by black rectangles, i.e. the example is composed of the set of multi-transfers $\{M_1, M_2\}$. The Petri net part of multi-transfers is graphically represented by plains arrows. Suppose that $M_1 = \langle \langle \mathcal{I}, \mathcal{O} \rangle, T \rangle$ The arrow from p_1 to M_1 and labelled by 1 in the MTN of Figure 2.2 means that $\mathcal{I}(p_1) = 1$. No plain transitions from p_2, p_3 and p_4 to M_1 mean that $\mathcal{I}(p_2) = 0$, $\mathcal{I}(p_3) = 0$ and $\mathcal{I}(p_4) = 0$. The arrow from M_1 to p_2 and labelled by 1 means that $\mathcal{O}(p_2) = 1$. No plain transitions from M_1 to p_1, p_3 and p_4 to M_1 mean that $\mathcal{O}(p_1) = 0$, $\mathcal{O}(p_3) = 0$ and $\mathcal{O}(p_4) = 0$. In the following, when $\mathcal{I}(p) = 1$, we will omit to label the corresponding arrow by 1. Dashed arrows from p_3 to M_1 and from M_1 to p_4 and the constant k_1 associated to M_1 mean that T is composed of one transfer from p_3 to p_4 with the bound k_1, i.e. $T = \{\langle \{\langle \{p_3\}, p_4 \rangle\}, k_1 \rangle\}$. When a multi-transfer deals with several transfer blocks, boxes are used to separate the places related to the different transfer blocks and different kinds of arrow are used to distinguish the different transfers into a transfer block (see Figure 2.5(f)).

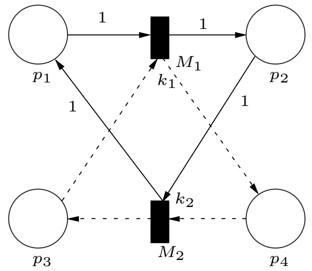

Figure 2.2: An example of multi-transfer net.

A *Petri net* [Rei86] (noted PN for short) is a MTN where all the multi-transfers have an empty set of transfer blocks.

A *Petri net with transfers* [Cia94] (noted PN + T for short) is a MTN where the bounds of all the transfer blocks of multi-transfers are equal to $+\infty$.

In the following, we will consider a new extension of Petri nets called *Petri net with non-blocking arcs*. A Petri net with non-blocking arcs (noted PN + NBA for short) is a MTN where the bounds of all the transfer blocks of multi-transfers are equal to 1.

A *labeled* MTN is a tuple $\langle \mathcal{P}, \mathcal{B}, \mathcal{L} \rangle$ where $\mathcal{P}$ and $\mathcal{B}$ are, respectively, a finite set of places and a finite set of multi-transfers and $\mathcal{L} : \mathcal{B} \to \Sigma$ is a labeling function that labels each multi-transfer $M \in \mathcal{B}$ with a label $\mathcal{L}(M)$ from the set of labels Σ.

A *marking* is a mapping $\mathbf{m} : \mathcal{P} \to \mathbb{N}$. In the following, we indistinctly consider markings, $\mathcal{I}$ and $\mathcal{O}$ (defining the input and output Petri net edges of multi-transfers) as functions from places to $\mathbb{N}$ or as tuples in $\mathbb{N}^{|\mathcal{P}|}$. The *component-wise ordering* $\preccurlyeq$ over tuples in $\mathbb{N}^k$ is defined as follows: $\mathbf{m} \preccurlyeq \mathbf{m}'$ if and only if $m_i \leq m'_i$ for $1 \leq i \leq k$. We denote $\mathbf{m} \prec \mathbf{m}'$ for $\mathbf{m} \preccurlyeq \mathbf{m}'$ but $\mathbf{m} \neq \mathbf{m}'$. Given $S \subseteq \mathcal{P}$ we define $\mathbf{m}(S) = \Sigma_{p \in S} \mathbf{m}(p)$. Then, we introduce the following definitions.

Definition 10 (Multi-transfer Enabled) Let $M = \langle \langle \mathcal{I}, \mathcal{O} \rangle, \{S_1, \ldots, S_u\} \rangle$ be a multi-transfer. We say that M is *enabled* in $\mathbf{m}$ if $\mathbf{m} \succcurlyeq \mathcal{I}$.

Definition 11 (Firing a Multi-transfer) Let $M = \langle \langle \mathcal{I}, \mathcal{O} \rangle, \{S_1, \ldots, S_u\} \rangle$ be a multi-transfer enabled in $\mathbf{m}$. *Firing* M in $\mathbf{m}$ leads to any marking $\mathbf{m}'$ (written $\mathbf{m} \rightarrowtail^{M} \mathbf{m}'$) computed in accordance to the following *sequence* of steps (in which we use the two intermediate markings $\mathbf{m}_1$ and $\mathbf{m}_2$):

1. Let $\mathbf{m}_1$ be such that $\mathbf{m}_1(p) = \mathbf{m}(p) - \mathcal{I}(p)$ for all $p \in \mathcal{P}$;

2. Let $\mathbf{m}_2$ be any marking such that the following constraints are satisfied:

   - for each transfer block $S_k = \langle \{B_{k1}, \ldots, B_{kr_k}\}, c_k \rangle$ of M:
     - if $\mathbf{m}_1(P_{k1} \cup \ldots \cup P_{kr_k}) > c_k$, then for all $d_1, \ldots, d_{r_k} \in \mathbb{N}$ be such that $d_1 + \ldots + d_{r_k} = c_k$ and for all j, $1 \leq j \leq r_k$:
       - $\mathbf{m}_2(p_{kj}) = \mathbf{m}_1(p_{kj}) + d_j$,
       - and $\mathbf{m}_2(P_{kj}) = \mathbf{m}_1(P_{kj}) - d_j$,
       - with the additional constraint that $\mathbf{m}_1(p) \geq \mathbf{m}_2(p)$ for any $p \in P_{kj}$.
     - if $\mathbf{m}_1(P_{k1} \cup \ldots \cup P_{kr_k}) \leq c_k$, then for all j, $1 \leq j \leq r_k$:
       - $\mathbf{m}_2(p_{kj}) = \mathbf{m}_1(p_{kj}) + \mathbf{m}_1(P_{kj})$ and
       - $\mathbf{m}_2(P_{kj}) = 0$;
   - $\mathbf{m}_2(p) = \mathbf{m}_1(p)$ for all $p \in \mathcal{P}$ not involved in transfer blocks;

3. then $\mathbf{m}'(p) = \mathbf{m}_2(p) + \mathcal{O}(p)$ for all $p \in \mathcal{P}$.

Intuitively, when the multi-transfer M_1 of Figure 2.2 is fired, one token is first removed from p_1. Then, if there is at least k_1 tokens into p_3, then k_1 tokens are moved from p_3 to p_4. Otherwise, all the tokens that are in p_3 are moved to p_4. Finally, one token is added into p_2.

Remark 1 *With a fixed bound $c \in \mathbb{N}$, a collection of transfer blocks of a multi-transfer may have a* non-deterministic *effect whenever the total number of tokens is greater than c. The non-determinism is due to the tokens that are transferred in each block, and from their distribution within the set P_{kj} of sources for $j : 1, \ldots, r_k$. For instance, if there is one token into a place p_1 and one token into a place p_2, then the transfer block $\langle\{\langle\{p_1\}, p_3\rangle\langle\{p_2\}, p_4\rangle\}, 1\rangle$ either moves one token from p_1 to p_3 or moves one token from p_2 to p_4.*

The operational semantics of a MTN is defined below. First, let us define the notion of transition system.

Definition 12 (Transition system) *A* transition system *is a tuple $\langle S, \rightarrow\rangle$ where S is a (possibly infinite) set of states and $\rightarrow \subseteq S \times S$ is a transition relation.*

Given a partial order $\sqsubseteq$, a transition system $\langle S, \rightarrow\rangle$ is *monotonic* with respect to $\sqsubseteq$ on S if for all s_1, s_2 in S with $s_1 \sqsubseteq s_2$, if $\langle s_1, s_1'\rangle \in\rightarrow$ then there exists s_2' with $s_1' \sqsubseteq s_2'$ and $\langle s_2, s_2'\rangle \in\rightarrow$ A transition system $\langle S, \rightarrow\rangle$ is *strictly monotonic* with respect to the partial order $\sqsubseteq$ on S if it is monotonic and for all s_1, s_2 in S with $s_1 \sqsubseteq s_2$ but $s_1 \neq s_2$, if $\langle s_1, s_1'\rangle \in\rightarrow$ then there exists s_2' with $s_1' \sqsubseteq s_2'$ but $s_2' \neq s_1'$ and $\langle s_2, s_2'\rangle \in\rightarrow$.

The transition system defined by a MTN is defined as follows.

Definition 13 (Transition system defined by a MTN) *The* transition system *defined by the* MTN *$\mathcal{G} = \langle\mathcal{P}, \mathcal{B}\rangle$ is the pair $\langle S, \rightarrow\rangle$ where $S = \mathbb{N}^{|\mathcal{P}|}$ and $\rightarrow \subseteq S \times S$ such that we have $\langle \mathbf{m}, \mathbf{m}'\rangle \in\rightarrow$ if and only if $\mathbf{m} \rightarrowtail^M \mathbf{m}'$ for some $M \in \mathcal{B}$.*

A (finite) *run* of the MTN $\mathcal{M}$ that defines the transition relation $\langle S, \rightarrow\rangle$ is a finite sequence of markings $\mathbf{m}_0\mathbf{m}_1 \ldots \mathbf{m}_n$ such that for any i, $0 \leq i < n$, we have $\langle \mathbf{m}_i, \mathbf{m}_{i+1}\rangle \in\rightarrow$, $\mathbf{m}_0$ is the *initial* marking of the run and $\mathbf{m}_n$ the *target* marking of the run. A marking $\mathbf{m}'$ is *reachable* from a marking $\mathbf{m}$, written $\mathbf{m} \rightarrowtail^* \mathbf{m}'$, if and only if there exists a run with initial marking $\mathbf{m}$ and target marking $\mathbf{m}'$. The *set of reachable markings* of $\mathcal{M}$ from a set of markings I, written $\mathsf{Reach}_{\mathcal{M}}(I)$, is defined as the set $\{\mathbf{m}' \mid \exists \mathbf{m} \in I : \mathbf{m} \rightarrowtail^* \mathbf{m}'\}$. Finite runs are extended to infinite runs in the obvious way.

For a labelled MTN $\mathcal{M}$ and a marking $\mathbf{m}$, we define $\mathcal{L}(\mathcal{M}, \mathbf{m}) = \{\eta_1\eta_2 \ldots \eta_n \ldots \mid$ we have $\mathbf{m}_0 \rightarrow^{M_1} \mathbf{m}_1 \rightarrowtail^{M_2} \ldots \rightarrowtail^{M_n} \mathbf{m}_n \ldots$ in $\mathcal{M}$ with $\mathbf{m}_0 = \mathbf{m}$ and $\mathcal{L}(M_i) = \eta_i$ for all $i \geq 1\}$.

To simplify the presentation, we will consider in the following that a place p is the target of at most one transfer in a transfer block. This last constraint ensures that a transfer $\langle P_1 \cup P_2, p\rangle$ is never decomposed into two (equivalent) transfers $\langle P_1, p\rangle$ and $\langle P_2, p\rangle$.

2.2.2 Properties of MTNs

In parameterized verification, Petri nets have been used to model several kinds of systems. In [GS92], Petri nets are used to describe the behaviours of parameterized systems where components communicate through rendez-vous. Petri nets are also used in [BCR01] to describe the behaviours of parameterized systems where components communicate using global variables instead of rendez-vous. When the communication mechanisms are more complex, extensions of Petri nets are needed. In [EN98, Del00], Petri nets extended with transfers are used to model systems communicating by broadcast.

No extension of Petri nets have been proposed to model systems where components communicate with signals. This means components communicating by sending a signal in the system. Either at least one component waits for this signal and the signal is caught by a component that can then react, or no component waits for the signal. In the last case, the signal is lost and could not be used afterwards.

Such systems can be modeled using MTNs where the bounds of transfer blocks are always equal to 1. This is particularly useful to describe the behaviours of JAVA programs with an unbounded number of threads communicating with the notify primitive (see chapter 3).

In [DFS98], some classical problems on Petri nets are studied on extended Petri nets. These problems are defined as follows.

The *marking reachability problem* for a Petri net P, is the problem defined as follows: "Given a Petri net P with an initial marking $\mathbf{m}$ and a marking $\mathbf{m}'$, does $\mathbf{m}'$ belong to $\mathsf{Reach}_P(\mathbf{m})$?".

The *marking coverability problem* for a Petri net P is the problem defined as follows: "Given a Petri net P with an initial marking $\mathbf{m}$ and a marking $\mathbf{m}'$, does there exist a marking $\mathbf{m}''$ that belongs to $\mathsf{Reach}_P(\mathbf{m})$ and such that $\mathbf{m}' \preccurlyeq \mathbf{m}''$?".

The *boundedness problem* for a Petri net P is the problem defined as follows : "Given a Petri net P and an initial marking $\mathbf{m}$, is $\mathsf{Reach}_P(\mathbf{m})$ finite ?".

The *place boundedness problem* for a Petri net P is the problem defined as follows : "Given a Petri net P, an initial marking m and a place p, is there $c \in \mathbb{N}$ such that $\forall \mathbf{m}' \in \mathsf{Reach}_P(\mathbf{m})$ we have $\mathbf{m}'(p) \leq c$?"

We also consider a further property called (action based) LTL model checking.

The formula of the (action based) LTL logic are evaluated over sequences of labels. Given a set of labels Σ, the formulas of the (action based) LTL logic are defined by the following rule:

$$\phi := \lambda | \neg\phi | \phi_1 \vee \phi_2 | \bigcirc \phi | \Box\phi | \Diamond\phi | \phi_1 \mathcal{U} \phi_2$$

where $\lambda \in \Sigma$. We only give the semantics for the $\Box$, $\Diamond$ and $\neg$ operators because they are the only one that we need in this thesis. For $\Lambda \in \Sigma^\omega$ such that $\Lambda = \lambda_1 \ldots \lambda_i \lambda_{i+1} \ldots$, we denote by Λ^i the suffix $\lambda_i \lambda_{i+1} \ldots$ of Λ starting at the index i, and $\Lambda(i) = \lambda_i$ (in both cases $i \geq 1$). Given $\Lambda \in \Sigma^\omega$ and a formula ϕ, we define the satisfaction relation, noted $\models$, as follows :

- if $\phi = \lambda$, then $\Lambda \models \phi$ iff $\Lambda(1) = \lambda$;
- if $\phi = \neg\varphi$, then $\Lambda \models \phi$ iff we do not have that $\Lambda \models \varphi$;
- if $\phi = \Diamond\varphi$, then $\Lambda \models \phi$ iff $\exists i \geq 1 : \Lambda^i \models \varphi$;
- if $\phi = \Box\varphi$, then $\Lambda \models \phi$ iff $\forall i \geq 1 : \Lambda^i \models \varphi$.

We denote that Λ does not satisfies ϕ by $\Lambda \not\models \phi$. For a set of infinite sequences of labels R and a formula ϕ, we have $R \models \phi$ if for all $\Lambda \in R$ we have $\Lambda \models \phi$, and $R \not\models \phi$ if there exists $\Lambda \in R$ such that $\Lambda \not\models \phi$.

Now we are ready to present the last property that we will consider.

The *action-based* LTL *model checking problem* for a labeled Petri net P is the problem defined as follows : "Given a labeled Petri net P, an initial marking $\mathbf{m}$ and an action-based LTL formula ϕ, does $\mathcal{L}(P, \mathbf{m}) \models \phi$ hold ?"

The previous definitions are extended to extensions of Petri nets in the obvious way.

Decidability and undecidability results for these problems on PN and $\mathsf{PN}+\mathsf{T}$ are shown on Table 2.1. For each result, we also give the reference to the paper that presents it [3]. In this section, we study the decidability of these classical problems on MTN.

Model	Coverability	Boundedness	Place boundedness	LTL Model checking	Reachability
PN	D [KM69]	D [KM69]	D [KM69]	D [Esp94]	D [Kos82, May84]
$\mathsf{PN}+\mathsf{T}$	D [DFS98]	D [DFS98]	I [DFS98]	I	I [DFS98]
MTN	?	?	?	?	?

Table 2.1: Decidability (D)/undecidability (I) results of classical problem on (extended) Petri nets.

To prove that some of these problems are undecidable, we present a reduction from a well-known undecidable problem on two counter machines.

Definition 14 A *two-counter machine* C, $\mathsf{2CM}$ for short, is a tuple $\langle c_1, c_2, L, \mathsf{Instr}\rangle$ where:

- c_1, c_2 are two counters taking their values in $\mathbb{N}$;
- $L = \{l_1, l_2, \ldots, l_u\}$ is a finite non-empty set of u locations;
- Instr is a function that labels each location $l \in L$ with an instruction that has one of the three following forms:
  - $l : c_j := c_j + 1;$ $\texttt{goto}$ l';, where $j \in \{1,2\}$ and $l' \in L$, this is called an increment, and we define $\mathsf{TypeInst}(l) = \mathtt{inc_j}$;
  - $l : c_j := c_j - 1;$ $\texttt{goto}$ l';, where $j \in \{1,2\}$ and $l' \in L$, this is called a decrement, and we define $\mathsf{TypeInst}(l) = \mathtt{dec_j}$;
  - l : $\texttt{if}$ $c_j = 0$ $\texttt{then goto}$ l' $\texttt{else goto}$ l'';, where $j \in \{1,2\}$ and $l', l'' \in L$, this is called a zero-test, and we define $\mathsf{TypeInst}(l) = \mathtt{zerotest_j}$.

Those instructions have their usual obvious semantics; in particular, decrement can only be done if the value of the counter is strictly greater than zero.

Given a $\mathsf{2CM}$ $C = \langle c_1, c_2, L, \mathsf{Instr}\rangle$, a *configuration* of C is a tuple $\langle loc, v^1, v^2\rangle$ where $loc \in L$ is the current location (=value of the program counter) and v^1, respectively v^2, is a natural number that gives the valuation of the counter c_1, respectively c_2. A *computation* γ of C is either a finite sequence of configurations $\langle loc_1, v_1^1, v_1^2\rangle, \langle loc_2, v_2^1, v_2^2\rangle, \ldots, \langle loc_r, v_r^1, v_r^2\rangle$, or an infinite sequence of configurations $\langle loc_1, v_1^1, v_1^2\rangle, \langle loc_2, v_2^1, v_2^2\rangle$ $, \ldots, \langle loc_r, v_r^1, v_r^2\rangle, \ldots$ such that: (i) "Initialization": $loc_1 = l_1$, $v_1^1 = 0$, and $v_1^2 = 0$, i.e. a computation starts in l_1 and the two counters have the value zero; (ii) "Consecution": for each $i \in \mathbb{N}$ with $1 < i \leq |\gamma|$ we have that $\langle loc_i, v_i^1, v_i^2\rangle$ is the configuration obtained from $\langle loc_{i-1}, v_{i-1}^1, v_{i-1}^2\rangle$ by applying the instruction $\mathsf{Instr}(loc_{i-1})$. In the finite case, r is the *length* of the computation γ and we define $\mathsf{final}(\gamma) = \langle loc_r, v_r^1, v_r^2\rangle$. If γ is a computation and $i \leq |\gamma|$, γ_i denotes the *ith* configuration of γ. A configuration $\langle loc, v^1, v^2\rangle$ is *reachable* in C if there exists a finite computation γ such that $\mathsf{final}(\gamma) = \langle loc, v^1, v^2\rangle$.

[3] The undecidability of LTL model checking on Petri nets with transfers cannot be directly found in the literature. However, the result can be easily obtained from the undecidability of liveness properties on broadcast protocols presented in [EFM99].

Problems. Here are two well-known undecidable problems on 2CM.

The reachability problem for 2CM
Instance: a 2CM $C = \langle c_1, c_2, L, \mathsf{Instr}\rangle$ and a configuration $\langle loc, v^1, v^2\rangle$ of C.
Question: Is $\langle loc, v^1, v^2\rangle$ reachable from $\langle l_1, 0, 0\rangle$?

The boundedness problem for 2CM
Instance: a 2CM $C = \langle c_1, c_2, L, \mathsf{Instr}\rangle$.
Question: Is there $c \in \mathbb{N}$ such that for all reachable configuration $\langle loc, v_1, v_2\rangle$ in C we have $v_1 + v_2 \leq c$?

Theorem 1 (From [Min67]) The reachability and boundedness problems are undecidable for 2CM.

Decidability Results

We give here two positive results for the analysis of MTNs. They are a direct consequence of the strict monotonicity property of this class of extended Petri nets.

Proposition 1 All the transition systems defined by a MTN $\mathcal{M} = \langle \mathcal{P}, \mathcal{B}\rangle$ with $k =| \mathcal{P} |$ are strictly monotonic with respect to the component-wise ordering over tuples in $\mathbb{N}^k$.

Proof. We prove that for two markings $\mathbf{m}_1$ and $\mathbf{m}_2$, if $\mathbf{m}_2 \succcurlyeq \mathbf{m}_1$, then for all $M \in \mathcal{B}$ such that $\mathbf{m}_1 \rightarrowtail^M \mathbf{m}'_1$ there exists $\mathbf{m}'_2$ such that $\mathbf{m}_2 \rightarrowtail^M \mathbf{m}'_2$ with $\mathbf{m}'_2 \succcurlyeq \mathbf{m}'_1$. Moreover, if $\mathbf{m}_1 \prec \mathbf{m}_2$, then $\mathbf{m}'_1 \prec \mathbf{m}'_2$.

Given $\mathbf{m}_1 \preccurlyeq \mathbf{m}_2$ with $\mathbf{m}_1 \rightarrowtail^M \mathbf{m}'_1$, let us construct $\mathbf{m}'_2$ following Definition 11. We use the intermediate markings $\mathbf{n}_1, \mathbf{n}'_1, \mathbf{n}_2, \mathbf{n}'_2$.
Let $M = \langle\langle \mathcal{I}, \mathcal{O}\rangle, \{S_1, \ldots, S_u\}\rangle$.

Step 1

We first remove all tokens removed by Petri net arcs, i.e. $\mathbf{n}_1 = \mathbf{m}_1 - \mathcal{I}$ and $\mathbf{n}_2 = \mathbf{m}_2 - \mathcal{I}$. As $\mathbf{m}_1 \preccurlyeq \mathbf{m}_2$, if follows that we have $\mathbf{n}_1 \preccurlyeq \mathbf{n}_2$. Moreover, if $\mathbf{m}_1 \prec \mathbf{m}_2$, then $\mathbf{n}_1 \prec \mathbf{n}_2$.

Step 2

Suppose that the application of the transfer blocks on $\mathbf{n}_1$ leads to $\mathbf{n}'_1$. Let us show how to construct $\mathbf{n}'_2$, the result of the application of the transfer blocks on $\mathbf{n}_2$. Remember that the sets of places involved in the transfer blocks of a multi-transfer are all pairwise disjoint (see Definition 9). For this reason, we treat each transfer block separately. For every transfer block $S_i = \langle\{B_{i_1}, \ldots, B_{i_{r_i}}\}, c\rangle$ with $B_{i_j} = \langle P_{i_j}, p_{i_j}\rangle$, we have the following. When applying S_i on $\mathbf{n}_2$, the idea is to remove at least the same number of tokens than when S_i is applied on $\mathbf{n}_1$. We remove more tokens from a place p when applying S_i on $\mathbf{n}_2$ only when there are less than c tokens in the source places of the transfer block in $\mathbf{n}_1$. In this case, we are sure that p contains zero token after the application of S_i from $\mathbf{n}_1$, i.e. $\mathbf{n}'_1(p) = 0$. This implies that, even if we remove more tokens from p when applying S_i from $\mathbf{n}_2$, we will have $\mathbf{n}'_1(p) \leq \mathbf{n}'_2(p)$.

More precisely, we consider the two following cases.

1. $\mathbf{n}_1(\bigcup_{j=1\ldots r_i} P_{i_j}) \leq c$. Then, $\mathbf{n}_1'(p_{i_j}) = \mathbf{n}_1(p_{i_j}) + \mathbf{n}_1(P_{i_j})$ and $\mathbf{n}_1'(P_{i_j}) = 0$ for all $1 \leq j \leq r_i$.

   We now look at $\mathbf{n}_2$. We have two cases.

   - $\mathbf{n}_2(\bigcup_{j=1\ldots r_i} P_{i_j}) \leq c$. Then, we apply the transfer block on $\mathbf{n}_2'$ as defined in definition 11 : $\mathbf{n}_2'(p_{i_j}) = \mathbf{n}_2(p_{i_j}) + \mathbf{n}_2(P_{i_j})$ and $\mathbf{n}_2'(P_{i_j}) = 0$ for all $1 \leq j \leq, r_i$.
     In this case, we have $\mathbf{n}_1'(p) = \mathbf{n}_2'(p) = 0$ for all $p \in \bigcup_{j=\ldots r_i} P_{i_j}$. Moreover, as $\mathbf{n}_1 \preccurlyeq \mathbf{n}_2$, we have $\mathbf{n}_1'(p_{i_j}) \leq \mathbf{n}_2'(p_{i_j})$ for all $1 \leq j \leq r_i$;
   - $\mathbf{n}_2(\bigcup_{j=1\ldots r_i} P_{i_j}) > c$. By definition 11, we must choose only c tokens among $\bigcup j = 1 \ldots r_i P_{i_j}$. We choose $d_1, \ldots, d_{r_i}$ such that $\sum_{j=1\ldots r_i} d_j = c$ and $\mathbf{n}_2(P_{i_j}) \geq d_j \geq \mathbf{n}_1(P_{i_j})$ for all $1 \leq j \leq r_i$. As $\mathbf{n}_1(\bigcup_{j=1 \leq r_i} P_{i_j}) \leq c \leq \mathbf{n}_2(\bigcup_{j=1\ldots r_i} P_{i_j})$, it is possible.
     Then, we define $\mathbf{n}_2'$ for the places in $\bigcup_{j=1\ldots r_i}(P_{i_j} \cup \{p_{i_j}\})$ as follows: for all $1 \leq j \leq r_i$ we have $\mathbf{n}_2'(p_{i_j}) = \mathbf{n}_2(p_{i_j}) + d_j$ and $\mathbf{n}_2'(P_{i_j}) = \mathbf{n}_2(P_{i_j}) - d_j$ with $\mathbf{n}_2(p) \geq \mathbf{n}_2'(p)$ for $p \in P_{i_j}$.
     As $\mathbf{n}_1'(\bigcup_{j=1\ldots r_i} P_{i_j}) = 0$, we have $\mathbf{n}_2'(p) \geq \mathbf{n}_1'(p)$, $\forall p \in \bigcup_j P_{i_j}$. Moreover, as we choose the d_j's such that $d_j \geq \mathbf{n}_1(P_{i_j})$ for all $1 \leq j \leq r_i$, we have $\mathbf{n}_2'(p_{i_j}) \geq \mathbf{n}_1'(p_{i_j}), \forall 1 \leq j \leq r_i$;

2. $\mathbf{n}_1(\bigcup_{j=1\ldots r_i} P_{i_j}) > c$. As $\mathbf{n}_2 \succcurlyeq \mathbf{n}_1$, we also have $\mathbf{n}_2(\bigcup_{j=1\ldots r_i} P_{i_j}) > c$. In this case, we perform *exactly* the same transfers of tokens on $\mathbf{n}_2$ than on $\mathbf{n}_1$. As $\mathbf{m}_1 \rightarrowtail^M \mathbf{m}_1'$, we know there exists $d_1, \ldots, d_{r_i}$ with $\sum_{j=1\ldots r_i} d_j = c$ such that we have the following. For all $1 \leq j \leq r_i$, let $\mathbf{n}_1'(p_{i_j}) = \mathbf{n}_1(p_{i_j}) + d_j$ and $\mathbf{n}_1'(P_{i_j}) = \mathbf{n}_1(P_{i_j}) - d_j$ with $\mathbf{n}_1(p) \geq \mathbf{n}_1'(p)$ for all $p \in P_{i_j}$.

   We define $\mathbf{n}_2'$ for these places as follows: for all $1 \leq j \leq r_i$ we have $\mathbf{n}_2'(p_{i_j}) = \mathbf{n}_2(p_{i_j}) + d_j$ and $\mathbf{n}_2'(p) = \mathbf{n}_2(p) - \mathbf{n}_1(p)$ for $p \in P_{i_j}$.

   In other words, we remove from $\mathbf{n}_2$ the same number of tokens than we removed from $\mathbf{n}_1$ for every place $p \in P_{i_j}$ and we add the same number of tokens to the places p_{i_j} $(1 \leq j \leq r_i)$ than we added to $\mathbf{n}_1$.

In all the cases, we have that $\mathbf{n}_1'(p) \leq \mathbf{n}_2'(p)$ for all $p \in \bigcup_{j=1\ldots r_i}(P_{i_j} \cup \{p_{i_j}\})$. Moreover, for all the other places, the marking remains the same, i.e. $\mathbf{n}_1'(p) = \mathbf{n}_1(p)$ and $\mathbf{n}_2'(p) = \mathbf{n}_2(p)$ for all $p \notin \bigcup_{j=1\ldots r_i}(P_{i_j} \cup \{p_{i_j}\})$.

As $\mathbf{n}_1 \preccurlyeq \mathbf{n}_2$, we conclude that $\mathbf{n}_1' \preccurlyeq \mathbf{n}_2'$. Moreover, suppose that $\mathbf{n}_1 \prec \mathbf{n}_2$. Two cases holds:

1. there is $p \notin \bigcup_{j=1\ldots r_i}(P_{i_j} \cup \{p_{i_j}\})$ such that $\mathbf{n}_1(p) < \mathbf{n}_2(p)$. As the marking does not change on this place, we conclude that $\mathbf{n}_1'(p) < \mathbf{n}_2'(p)$, and hence $\mathbf{n}_1' \prec \mathbf{n}_2'$;

2. there is $p \in \bigcup_{j=1\ldots r_i}(P_{i_j} \cup \{p_{i_j}\})$ with $\mathbf{n}_1(p) < \mathbf{n}_2(p)$. Suppose that $p \in P_{i_j} \cup \{p_{i_j}\}$ $(1 \leq j \leq r_i)$. Hence we have $\mathbf{n}_1(P_{i_j} \cup \{p_{i_j}\}) < \mathbf{n}_2(P_{i_j} \cup \{p_{i_j}\})$ as $\mathbf{n}_1 \preccurlyeq \mathbf{n}_2$. Moreover, as we have (i) $\mathbf{n}_1(P_{i_j} \cup \{p_{i_j}\}) = \mathbf{n}_1'(P_{i_j} \cup \{p_{i_j}\})$ and (ii) $\mathbf{n}_2(P_{i_j} \cup \{p_{i_j}\}) = \mathbf{n}_2'(P_{i_j} \cup \{p_{i_j}\})$ (no tokens are lost and no tokens are created when applying a transfer block), we conclude that there exists $p' \in \bigcup_{j=1\ldots r_i}(P_{i_j} \cup \{p_{i_j}\})$ such that $\mathbf{n}_1'(p') < \mathbf{n}_2'(p')$.

In both cases, we have that if $\mathbf{n}_1 \prec \mathbf{n}_2$, then $\mathbf{n}_1' \prec \mathbf{n}_2'$.

Step 3

We add tokens with respect to $\mathcal{O}$. Following Definition 11, we have $\mathbf{m}'_1 = \mathbf{n}'_1 + \mathcal{O}$ and $\mathbf{m}'_2 = \mathbf{n}'_2 + \mathcal{O}$. As $\mathbf{n}'_1 \preccurlyeq \mathbf{n}'_2$, then $\mathbf{m}'_1 \preccurlyeq \mathbf{m}'_2$. Moreover, if $\mathbf{n}'_1 \prec \mathbf{n}'_2$ then we have $\mathbf{m}'_1 \prec \mathbf{m}'_2$.

Finally, we obtain that $\mathbf{m}_2 \rightarrowtail^M \mathbf{m}'_2$ and $\mathbf{m}'_2 \succcurlyeq \mathbf{m}'_1$. Moreover, if $\mathbf{m}_1 \prec \mathbf{m}_2$, then $\mathbf{m}'_1 \prec \mathbf{m}'_2$. We conclude that MTNs describes transition systems that are strictly monotonic. □

In [ACJT96, FS01], algorithms for the verification of the coverability and boundedness problems are presented for (strictly) monotonic systems. From Proposition 1 and [ACJT96, FS01], we deduce the decidability of the coverability problem and the boundedness problem for MTNs.

Corollary 1 The coverability problem and the boundedness problem are decidable for MTNs.

Undecidability Results In the previous section, we have seen that the coverability problem and the boundedness problem are decidable for MTNs.

Since the class of PN + T is a subclass of MTN and the marking reachability, action-based LTL model checking and place boundedness problems are undecidable for PN + T (see Table 2.1), we directly obtain the undecidability of those problems for MTNs.

Corollary 2 The marking reachability, action-based LTL model checking and place boundedness problems are undecidable for MTN.

In this section we show that those results still hold for the restricted subclass PN + NBA of MTN (where all the bounds are equal to 1) that does not contain the class PN + T (where all the bounds are equal to $+\infty$) and that allows us to model systems where components communicate by sending signals (see Chapter 3).

To establish the undecidability results, we will show that PN + NBA are able to partially simulate the computations of a 2CM. This partial simulation result will allow us to reduce in a uniform way undecidable problems for 2CM to problems for PN + NBA.

Partial simulations of a 2CM by a PN + NBA First, we present a widget, i.e. a construction that will be used in all the proofs.

Widget. For any 2CM $C = \langle c_1, c_2, L = \{l_1, l_2, \ldots, l_u\}, \mathsf{Instr}\rangle$, we construct a PN + NBA $\mathcal{N}_C = \langle \mathcal{P}, \mathcal{B}\rangle$, called the *simulation widget*, defined as follows. The set of places $\mathcal{P}$ is equal to $\{c_1, c_2, l_1, l_2, \ldots, l_u, K, T\}$. The places c_1 and c_2 will be used to keep track of the values of the two counters of C, $l_1, l_2, \cdots, l_u$ called the *control places* will be used to keep track of the program counter of C, K is called the *capacity place*, T is called the *trash*. The use of K and T will be described below. The set of multi-transfers $\mathcal{B}$ is the smallest set of multi-transfers such that for each $l_i \in L$:

- if $\mathsf{Instr}(l_i)$ is of the form $c_j := c_j + 1; \texttt{goto}\ l'$, then $\mathcal{B}$ contains the multi-transfer $M_i = \langle\langle \mathcal{I}, \mathcal{O}\rangle, B\rangle$ with $\mathcal{I}(p) = 1$ if $p \in \{l_i, K\}$, 0 otherwise; $\mathcal{O}(p) = 1$ if $p \in \{c_j, l'\}$, 0 otherwise; and $B = \emptyset$;
- if $\mathsf{Instr}(l_i)$ is of the form $c_j := c_j - 1; \texttt{goto}\ l'$, then $\mathcal{B}$ contains the multi-transfer $M_i = \langle\langle \mathcal{I}, \mathcal{O}\rangle, B\rangle$ with $\mathcal{I}(p) = 1$ if $p \in \{l_i, c_j\}$, 0 otherwise; $\mathcal{O}(p) = 1$ if $p \in \{l', K\}$, 0 otherwise; and $B = \emptyset$;

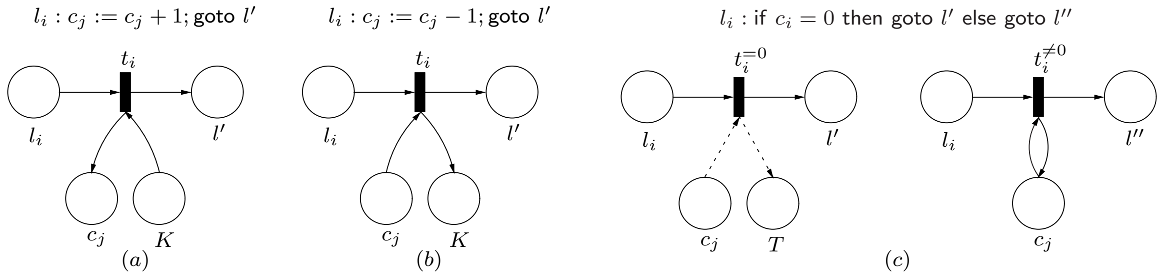

Figure 2.3: Simulation of the operations of a 2CM by PN + NBA multi-transfers.

- if $\mathsf{Instr}(l_i)$ is of the form `if` $c_j = 0$ `then goto` l' `else goto` l'' then $\mathcal{B}$ contains two multi-transfers $M_i^{=0}$ and $M_i^{\neq 0}$ defined as:
  - $M_i^{=0} = \langle\langle \mathcal{I}, \mathcal{O}\rangle, B\rangle$ with $\mathcal{I}(p) = 1$ if $p = l_i$, 0 otherwise; $\mathcal{O}(p) = 1$ if $p = l'$, 0 otherwise and $B = \{\langle\langle\{c_j\}, T\rangle, 1\rangle\}$;
  - $M_i^{\neq 0} = \langle\langle \mathcal{I}, \mathcal{O}\rangle, B\rangle$ with $\mathcal{I}(p) = 1$ if $p \in \{l_i, c_j\}$, 0 otherwise; $\mathcal{O}(p) = 1$ if $p \in \{c_j, l''\}$, 0 otherwise and $B = \emptyset$.

Figure 2.3(a) shows the multi-transfer that simulates an increment of c_j by moving one token from the capacity place to c_j. Figure 2.3(b) shows the multi-transfer that simulates a decrement of c_j by moving one token from c_j to the capacity place. Figure 2.3(c) shows the multi-transfers that simulates a zero-test on c_j when c_j is equal to zero (multi-transfer $M_i^{=0}$) and when c_j is greater than zero (multi-transfer $M^{\neq 0}$). We note $\mathbf{m}_k$ the (initial) marking of the places in $\mathcal{P} = \{c_1, c_2, l_1, l_2, \ldots, l_u, K, T\}$ defined as follows: $\mathbf{m}_k(l_1) = 1$, for any $l \in \{l_2, l_3, \ldots, l_u\}$, $\mathbf{m}_k(l) = 0$, $\mathbf{m}_k(c_1) = 0$, $\mathbf{m}_k(c_2) = 0$, $\mathbf{m}_k(K) = k$, and $\mathbf{m}_k(T) = 0$.

Properties of the widget. Let $C = \langle c_1, c_2, L, \mathsf{Instr}\rangle$ be a 2CM and $\mathcal{N}_C = \langle \mathcal{P}, \mathcal{B}\rangle$ be the simulation widget associated to C as defined above. Let $\gamma = \langle loc_1, v_1^1, v_1^2\rangle\langle loc_2, v_2^2, v_2^2\rangle \ldots$ be the computation of C. We associate to γ a sequence of multi-transfers $M_1 M_2 \ldots$ of $\mathcal{N}_C$, such that for all $i \in \mathbb{N}$ such that $1 \leq i \leq |\gamma|$, we have $M_i = \alpha(\langle loc_i, v_i^1, v_i^2\rangle)$ where α is defined as:

$$\alpha(\langle loc, v_1, v_2\rangle) = \begin{cases} M_k & \text{if } loc = l_k \text{ and } \mathsf{TypeInst}(loc) \neq \texttt{zerotest}_\texttt{j} \\ M_k^{=0} & \text{if } loc = l_k \text{ and } \mathsf{TypeInst}(loc) = \texttt{zerotest}_\texttt{j} \text{ and } v_j = 0. \\ M_k^{\neq 0} & \text{if } loc = l_k \text{ and } \mathsf{TypeInst}(loc) = \texttt{zerotest}_\texttt{j} \text{ and } v_j > 0. \end{cases}$$

The sequence of multi-transfers corresponding to γ is denoted by $\alpha(\gamma)$.

The function α^{-1} on sequences of multi-transfers $\sigma = M_1 \ldots M_n$ of the widget that are firable from $\mathbf{m}_k$ ($k \geq 1$) returns a sequence of configurations of C $\gamma = \langle loc_0, v_0^1, v_0^2\rangle$ $\langle loc_1, v_1^1, v_1^2\rangle \ldots \langle loc_n, v_n^1, v_n^2\rangle$ such that (i) $loc_0 = l_1$, $v_0^1 = 0$, $v_0^2 = 0$ and (ii) for all $1 \leq i \leq n$, either $\mathsf{TypeInstr}(l_{i-1}) \neq \mathit{zerotest}_j$ and $\langle loc_i, v_i^1, v_i^2\rangle$ is constructed from $\langle loc_{i-1}, v_{i-1}^1, v_{i-1}^2\rangle$ applying $\mathsf{Instr}(l_{i-1})$. Or $\mathsf{Instr}(l_{i-1})$ is of the form if $c_j = 0$ then goto l' else goto l'' and the following cases holds.

- $M_i = M^{=0}$, then $loc_i = l'$, $v_i^1 = v_{i-1}^1$ and $v_i^2 = v_{i-1}^2$, or
- $M_i = M^{\neq 0}$ and $loc_i = l''$, $v_i^1 = v_{i-1}^1$, $v_i^2 = v_{i-1}^2$.

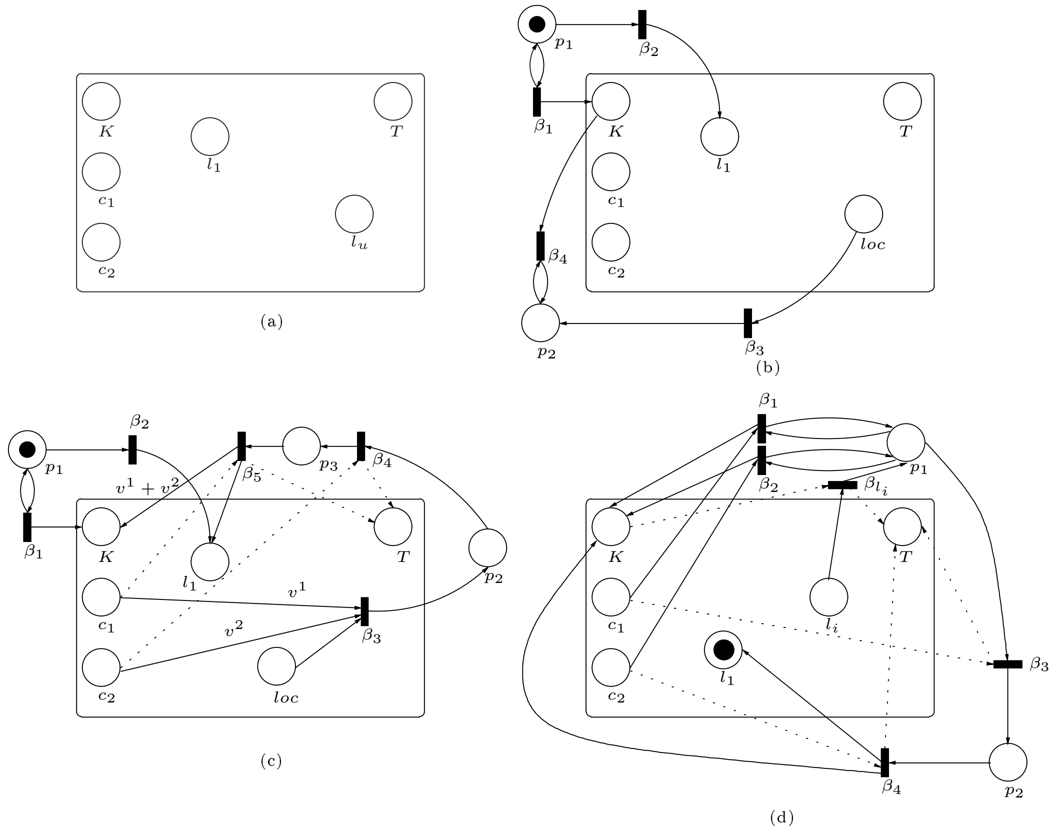

Figure 2.4: Constructions using the widget.

Notice that α^{-1} is a function: the result is unique. We now formalize important properties of the widget by the following lemmas.

Lemma 1 Let $\gamma = \langle loc_1, v_1^1, v_1^2\rangle\langle loc_2, v_2^1, v_2^2\rangle \dots \langle loc_r, v_r^1, v_r^2\rangle$ be a computation of the 2CM $C = \langle c_1, c_2, L, \mathsf{Instr}\rangle$ such that for any $i \in \{1, 2, \dots, r\}$, $v_i^1 + v_i^2 \leq k$. Let $\mathcal{N}_C$ be the simulation widget associated to C. The sequence of multi-transfers $\alpha(\gamma)$ is firable from the marking $\mathbf{m}_k$ and firing this sequence of multi-transfers leads to a marking $\mathbf{m}'$ defined as follows: $\mathbf{m}'(l) = 1$, for $l = loc_r$, $\mathbf{m}'(l') = 0$ for any $l' \neq loc_r$, $\mathbf{m}'(c_1) = v_r^1$, $\mathbf{m}'(c_2) = v_r^2$, $\mathbf{m}'(K) = k - v_r^1 - v_r^2$, and $\mathbf{m}'(T) = 0$.

Proof. By induction on the length of the computations of C (notice that each prefix of γ is a computation where the sum of the counters is less or equal to k). The basic case ($l = 0$) is obvious. Suppose that the lemma holds for all the computations of size $l < n$.

Let $\gamma = \gamma' \cdot \langle loc_n, v_n^1, v_n^2\rangle$ be the computation of the 2CM C of size n where $\gamma' = \langle loc_1, v_1^1, v_1^2\rangle \dots \langle loc_{n-1}, v_{n-1}^1, v_{n-1}^2\rangle$. By induction hypothesis, we have that $\alpha(\gamma')$ leads to the marking $\mathbf{m}'$ in $\mathcal{N}_C$ such that $\mathbf{m}'(l_i) = 1$ if $l_i = loc_{n-1}$, $\mathbf{m}'(l_i) = 0$ for all $l_i \in L \setminus \{loc_{n-1}\}$, $\mathbf{m}'(c_1) = v_{n-1}^1$, $\mathbf{m}'(c_2) = v_{n-1}^2$, $\mathbf{m}'(K) = k - v_{n-1}^1 - v_{n-1}^2$ and $\mathbf{m}'(T) = 0$. The following cases hold.

1. If $\mathsf{Instr}(loc_{n-1})$ is of the form $c_j := c_j + 1; \mathsf{goto}\ l'$, then $\alpha(\langle loc_{n-1}, v_{n-1}^1, v_{n-1}^2\rangle) = M$ such that $M = \langle \mathcal{I}, \mathcal{O}, \emptyset\rangle$ where $\mathcal{I}(p) = 1$ if $p \in \{loc_{n-1}, K\}$, 0 otherwise; and $\mathcal{O}(p) = 1$

if $p \in \{(l', c_j\}$, 0 otherwise. By definition, we have $v_n^j = v_{n-1}^j + 1$ and $v_n^{3-j} = v_{n-1}^{3-j}$. Moreover, we have $\mathbf{m}'(K) = k - v_{n-1}^1 - v_{n-2}^2$ by hypothesis. We conclude that we have $\mathbf{m}' \rightarrowtail^M \mathbf{m}''$ such that $\mathbf{m}''(l') = 1$, $\mathbf{m}''(l_i) = 0$ for all $l_i \in L \setminus \{l'\}$, $\mathbf{m}''(c_1) = v_n^1$, $\mathbf{m}''(c_2) = v_n^2$, $\mathbf{m}''(K) = k - v_n^1 - v_n^2$ and $\mathbf{m}''(T) = 0$.

2. If $\mathsf{Instr}(loc_{n-1})$ is of the form $c_j := c_j - 1; \mathsf{goto}\ l'$, then $\alpha(\langle loc_{n-1}, v_{n-1}^1, v_{n-1}^2\rangle) = M$ such that $M = \langle \mathcal{I}, \mathcal{O}, \emptyset\rangle$ where $\mathcal{I}(p) = 1$ if $p \in \{loc_{n-1}, c_j\}$, 0 otherwise; and $\mathcal{O}(p) = 1$ if $p \in \{l', K\}$, 0 otherwise. As $\langle loc_{n-1}, v_{n-1}^1, v_{n-1}^2\rangle$ has a successor, we have $v_{n-1}^j > 0$, $v_n^j = v_{n-1}^j$, $v_{n-1}^{3-j} = v_{n-1}^{3-j}$ and $\mathbf{m}' \rightarrowtail^M \mathbf{m}''$ such that $\mathbf{m}''(l') = 1$, $\mathbf{m}''(l_i) = 0$ for all $l_i \in L \setminus \{l'\}$, $\mathbf{m}''(c_1) = v_n^1$, $\mathbf{m}''(c_2) = v_n^2$, $\mathbf{m}''(K) = k - v_n^1 - v_n^2$ and $\mathbf{m}''(T) = 0$.

3. If $\mathsf{Instr}(loc_{n-1})$ is of the form $\mathsf{if}\ c_j = 0\ \mathsf{then\ goto}\ l'\ \mathsf{else\ goto}\ l''$, then if $v_{n-1}^j = 0$ we have $\alpha(\langle loc_{n-1}, v_{n-1}^1, v_{n-1}^2\rangle) = M^{=0}$ such that $M^{=0} = \langle \mathcal{I}^{=0}, \mathcal{O}^{=0}, \{\langle\langle\{c_j\}, T\rangle, 1\rangle\}\rangle$ where $\mathcal{I}^{=0}(p) = 1$ if $p = loc_{n-1}$, 0 otherwise; and $\mathcal{O}^{=0}(p) = 1$ if $p = l'$, 0 otherwise. $M^{=0}$ is firable from $\mathbf{m}'$ and no tokens are moved to T as $\mathbf{m}_{n-1}^j = v_{n-1}^j$ and $v_{n-1}^j = 0$. We conclude that we have $\mathbf{m}' \rightarrowtail^{M^{=0}} \mathbf{m}''$ such that $\mathbf{m}''(l') = 1$, $\mathbf{m}''(l_i) = 0$ for all $l_i \in L \setminus \{l'\}$, $\mathbf{m}''(c_1) = v_n^1 (= v_{n-1}^1)$, $\mathbf{m}''(c_2) = v_n^2 (= v_{n-1}^2)$, $\mathbf{m}''(K) = k - v_n^1 - v_n^2$ and $\mathbf{m}''(T) = 0$. Otherwise if $v_{n-1}^j > 0$ we have $\alpha(\langle loc_{n-1}, v_{n-1}^1, v_{n-1}^2\rangle) = M^{\neq 0}$ such that $M^{\neq 0} = \langle \mathcal{I}^{\neq 0}, \mathcal{O}^{\neq 0}, \emptyset\rangle$ where $\mathcal{I}^{\neq 0}(p) = 1$ if $p \in \{loc_{n-1}, c_j\}$, 0 otherwise; and $\mathcal{O}^{\neq 0}(p) = 1$ if $p \in \{l'', c_j\}$. $M^{\neq 0}$ is firable from $\mathbf{m}'$ and we have $\mathbf{m}' \rightarrowtail^{M^{\neq 0}} \mathbf{m}''$ such that $\mathbf{m}''(l'') = 1$, $\mathbf{m}''(l_i) = 0$ for all $l_i \in L \setminus \{l''\}$, $\mathbf{m}''(c_1) = v_n^1 (= v_{n-1}^1)$, $\mathbf{m}''(c_2) = v_n^2 (= v_{n-1}^2)$, $\mathbf{m}''(K) = k - v_n^1 - v_n^2$ and $\mathbf{m}''(T) = 0$. □

This lemma formalizes the fact that any computation of a 2CM on which the sum of counters does not exceed k, can be faithfully simulated by its associated widget from marking $\mathbf{m}_k$, with a computation that does not put any token in T.

Lemma 2 Let $\sigma = M_1 M_2 \ldots M_n$ be a sequence of multi-transfers of the simulation widget $\mathcal{N}_C$ associated to the 2CM $C = \langle c_1, c_2, L, \mathsf{Instr}\rangle$. If $\mathbf{m}_k \rightarrowtail^\sigma \mathbf{m}'$ and $\mathbf{m}'(T) = 0$, then $\alpha^{-1}(\sigma)$ is a computation of C with $\mathsf{final}(\alpha^{-1}(\sigma)) = \langle loc, v^1, v^2\rangle$ such that $\mathbf{m}'(loc) = 1$, $v^1 = \mathbf{m}'(c_1)$ and $v^2 = \mathbf{m}'(c_2)$.

Proof. By induction on the size of the sequence of multi-transfers (notice that each prefix of a firable sequence is firable). The basic case ($l = 0$) is obvious. Suppose that the lemma holds for all a sequences of multi-transfers of size $l < n$. Let $\sigma = \sigma' \cdot M_n$ be a sequence of multi-transfers of $\mathcal{N}_C$ of size n that is firable from $\mathbf{m}_k$ with $\mathbf{m}_k \rightarrowtail^\sigma \mathbf{m}$ and $\mathbf{m}(T) = 0$, and where $\sigma' = M_1 \ldots M_{n-1}$. Since σ is firable from $\mathbf{m}_k$ so is σ', and since the marking of T is equal to zero after firing σ, due to the fact that the marking of T may only increase, the same is true after σ'. By induction hypothesis and the previous property, we have that $\mathbf{m}_k \rightarrowtail^{\sigma'} \mathbf{m}'$ and $\mathsf{final}(\alpha^{-1}(\sigma')) = \langle loc, v_1^1, v_1^2\rangle$ such that $\mathbf{m}'(loc) = 1$, $\mathbf{m}'(c_1) = v_1^1$, $\mathbf{m}'(c_2) = v_1^2$ and $\mathbf{m}'(T) = 0$. The following cases holds.

1. if $\lambda(M_n) = loc$ and $\mathsf{Instr}(loc)$ is of the form $c_j := c_j + 1; \mathsf{goto}\ l'$, then $M_n = \langle \mathcal{I}, \mathcal{O}, \emptyset\rangle$ such that $\mathcal{I}(p) = 1$ if $p \in \{loc, K\}$, 0 otherwise; and $\mathcal{O}(p) = 1$ if $p \in \{l', c_j\}$, 0 otherwise. We then have $\mathbf{m}' \rightarrowtail^{M_n} \mathbf{m}''$ and $\alpha^{-1}(\sigma)$ is a computation of C with $\mathsf{final}(\alpha^{-1}(\sigma)) = \langle l', v_2^1, v_2^2\rangle$ such that $\mathbf{m}''(l') = 1$, $\mathbf{m}''(c_1) = v_2^1$, $\mathbf{m}''(c_2) = v_2^2$ and $\mathbf{m}''(T) = 0$.

2. if $\lambda(tr_n) = loc$ and $\mathsf{Instr}(loc)$ is of the form $c_j := c_j - 1$; $\mathsf{goto}\ l'$, then $M_n = \langle \mathcal{I}, \mathcal{O}, \emptyset \rangle$ such that $\mathcal{I}(p) = 1$ if $p \in \{loc, c_j\}$, 0 otherwise; and $\mathcal{O}(p) = 1$ if $p \in \{l', K\}$, 0 otherwise. We then have $\mathbf{m}' \rightarrowtail^{M_n} \mathbf{m}''$ and $\alpha^{-1}(\sigma)$ is a computation of C with $\mathsf{final}(\alpha^{-1}(\sigma)) = \langle l', v_2^1, v_2^2 \rangle$ such that $\mathbf{m}''(l') = 1$, $\mathbf{m}''(c_1) = v_2^1$, $\mathbf{m}''(c_2) = v_2^2$ and $\mathbf{m}''(T) = 0$.

3. if $\mathsf{Instr}(\alpha^{-1}(M_n))$ is of the form $\mathsf{if}\ c_j = 0\ \mathsf{then\ goto}\ l'\ \mathsf{else\ goto}\ l''$, then if $\mathbf{m}'(c_j) = 0$, M_n must be such that $M_n = \langle \mathcal{I}, \mathcal{O}, \{\langle\langle \{c_j\}, T \rangle, 1 \rangle\} \rangle$ with $\mathcal{I}(p) = 1$ if $p = loc$, 0 otherwise; and $\mathcal{O}(p) = 1$ if $p = l'$, 0 otherwise. As $\mathbf{m}'(c_j) = 0$ the transfer $\langle \{c_j\}, T \rangle$ has no effect and we have $\mathbf{m}' \rightarrowtail^{M_n} \mathbf{m}''$ and $\alpha^{-1}(\sigma)$ is a computation of C with $\mathsf{final}(\alpha^{-1}(\sigma)) = \langle l', v_2^1, v_2^2 \rangle$ such that $\mathbf{m}''(l') = 1$, $\mathbf{m}''(c_1) = v_2^1$, $\mathbf{m}''(c_2) = v_2^2$ and $\mathbf{m}''(T) = 0$. Otherwise if $\mathbf{m}'(c_j) > 0$, M_n may not be of the kind $\langle \langle \mathcal{I}, \mathcal{O} \rangle, \{\langle \{\langle \{c_j\}, T \rangle\}, 1 \rangle\} \rangle$ since otherwise T would contain one token after firing M_n (since $\mathbf{m}'(c_j) \geq 1$, a transfer would indeed occur), hence M_n must be such that $M_n = \langle \mathcal{I}, \mathcal{O}, \emptyset \rangle$ with $\mathcal{I}(p) = 1$ if $p \in \{loc, c_j\}$, 0 otherwise; and $\mathcal{O}(p) = 1$ if $p \in \{l'', c_j\}$, 0 otherwise. If it is not the case, T would contain one token after firing M_n. We have $\mathbf{m}' \rightarrowtail^{M_n} \mathbf{m}''$ and $\alpha^{-1}(\sigma)$ is a computation of C with $\mathsf{final}(\alpha^{-1}(\sigma)) = \langle l'', v_2^1, v_2^2 \rangle$ such that $\mathbf{m}''(l'') = 1$, $\mathbf{m}''(c_1) = v_2^1$, $\mathbf{m}''(c_2) = v_2^2$ and $\mathbf{m}''(T) = 0$. □

This second lemma says that any computation of the widget from its initial marking that does not put tokens in T is a simulation of a computation of its associated 2CM.

Since the tokens that are in the trash place T always stay in T, this second lemma can be generalized for initial markings where there is already tokens in T.

Lemma 3 Let $\sigma = M_1 M_2 \ldots M_n$ be a sequence of multi-transfers of the simulation widget $\mathcal{N}_C$ associated to the 2CM $C = \langle c_1, c_2, L, \mathsf{Instr} \rangle$. Let $\mathbf{m}'_k$ be a marking such that $\mathbf{m}'_k(p) = \mathbf{m}_k(p)$ for all the places $p \neq T$ and $\mathbf{m}'_k(T) \geq 0$. If $\mathbf{m}'_k \rightarrowtail^{\sigma} \mathbf{m}'$ and $\mathbf{m}'(T) = \mathbf{m}'_k(T)$, then $\alpha^{-1}(\sigma)$ is a computation of C with $\mathsf{final}(\alpha^{-1}(\sigma)) = \langle loc, v^1, v^2 \rangle$ such that $\mathbf{m}'(loc) = 1$, $v^1 = \mathbf{m}'(c_1)$ and $v^2 = \mathbf{m}'(c_2)$.

To prove our undecidability results, we need the following last property of the widget.

Lemma 4 Let $\mathcal{N}_C$ be the simulation widget associated to the 2CM $C = \langle c_1, c_2, L, \mathsf{Instr} \rangle$. For any marking $\mathbf{m} \in \mathsf{Reach}_{\mathcal{N}_C}(\mathbf{m}_k)$, we have that $\mathbf{m}(\{c_1, c_2, K, T\}) = k$.

Proof. By induction on the size l of the minimal computation of $\mathcal{N}_C$ that allows us to reach $\mathbf{m}$. The basic case ($l = 0$) is obvious since $\mathbf{m}_k(c_1) = \mathbf{m}_k(c_2) = \mathbf{m}_k(T) = 0$ and $\mathbf{m}_k(K) = k$. Suppose that the lemma holds for all the markings reachable in i steps from $\mathbf{m}_k$ in $\mathcal{N}_C$ with $i < n$. Suppose that $\mathbf{m}'$ is reachable by firing $n - 1$ multi-transfers and we have $\mathbf{m}' \rightarrowtail^{M} \mathbf{m}$ for some multi-transfer M of $\mathcal{N}$. M can be of the following forms:

1. M corresponds to a decrement. More precisely, $M = \langle \mathcal{I}, \mathcal{O}, \emptyset \rangle$ with $\mathcal{I}(p) = 1$ if $p \in \{l, c_j\}$, 0 otherwise; and $\mathcal{O}(p) = 1$ if $p \in \{l', K\}$, 0 otherwise. In this case we have $\mathbf{m}(\{c_1, c_2, K, T\}) = \mathbf{m}'(\{c_1, c_2, K, T\})$.

2. M corresponds to an increment. More precisely, $M = \langle \mathcal{I}, \mathcal{O}, \emptyset \rangle$ with $\mathcal{I}(p) = 1$ if $p \in \{l, K\}$, 0 otherwise; and $\mathcal{O}(p) = 1$ if $p \in \{l', c_j\}$. In this case we have $\mathbf{m}(\{c_1, c_2, K, T\}) = \mathbf{m}'(\{c_1, c_2, K, T\})$.

3. M corresponds to a successful test for zero on c_j. More precisely, we have that $M = \langle \mathcal{I}, \mathcal{O}, \{\langle\langle\{c_j\}, T\rangle, 1\rangle\}\rangle$ with $\mathcal{I}(p) = 1$ if $p = l$, 0 otherwise; and $\mathcal{O}(p) = 1$ if $p = l''$, 0 otherwise. In this case, if $\mathbf{m}'(c_j) = 0$ as well as if $\mathbf{m}'(c_j) > 0$, we have $\mathbf{m}(\{c_1, c_2, K, T\}) = \mathbf{m}'(\{c_1, c_2, K, T\})$.

4. M corresponds to an unsuccessful test for zero on c_j when $\mathbf{m}'(c_j) > 0$. More precisely, $M = \langle \mathcal{I}, \mathcal{O}, \emptyset\rangle$ with $\mathcal{I}(p) = 1$ if $p \in \{l, c_j\}$, 0 otherwise; and $\mathcal{O}(p) = 1$ if $p \in \{l'', c_j\}$, 0 otherwise. Again we have $\mathbf{m}(\{c_1, c_2, K, T\}) = \mathbf{m}'(\{c_1, c_2, K, T\})$.

By induction hypothesis, we have $\mathbf{m}'(\{c_1, c_2, K, T\}) = \mathbf{m}_k(\{c_1, c_2, K, T\})$. So we conclude that $\mathbf{m}(\{c_1, c_2, K, T\}) = \mathbf{m}_k(\{c_1, c_2, K, T\})$. □

This last lemma says that in any reachable marking of the widget, the sum of the tokens in the set of places $\{c_1, c_2, K, T\}$ stays constant.

Again, this lemma can be generalised as follows.

Lemma 5 Let $\mathcal{N}_C$ be the simulation widget associated to the 2CM $C = \langle c_1, c_2, L, \mathsf{Instr}\rangle$ and $\mathbf{m}$ be a marking such that $\mathbf{m}(\{c_1, c_2, K, T\}) = k$. For any marking $\mathbf{m}' \in \mathsf{Reach}_{\mathcal{N}_C}(\mathbf{m})$, we have that $\mathbf{m}'(\{c_1, c_2, K, T\}) = k$.

Undecidability proofs We are now equipped to establish the undecidability of the marking reachability, action-based LTL model checking and place boundedness problems.

Theorem 2 The marking reachability problem is undecidable for $\mathsf{PN} + \mathsf{NBA}$.

Proof. Let $C = \langle c_1, c_2, L, \mathsf{Instr}\rangle$ be a 2CM and let $s = \langle loc, v^1, v^2\rangle$ be a configuration of C. Let us show that we can reduce the reachability problem of s in C to the reachability problem between two markings in a $\mathsf{PN} + \mathsf{NBA}$. The property will result from the undecidability of the reachability problem of 2CM.

We construct the $\mathsf{PN} + \mathsf{NBA}$ $\mathcal{N}' = \langle \mathcal{P}', \mathcal{B}'\rangle$ starting from the simulation widget $\mathcal{N}_C = \langle \mathcal{P}, \mathcal{B}\rangle$ associated to C. To the simulation widget schematized in Figure 2.4(a), we add places and multi-transfers as indicated in Figure 2.4(b). That is, $\mathcal{P}' = \mathcal{P} \cup \{p_1, p_2\}$, $\mathcal{B}' = \mathcal{B} \cup \{\beta_1, \beta_2, \beta_3, \beta_4\}$ and the new multi-transfers are defined as follows: $\beta_1 = \langle \mathcal{I}, \mathcal{O}, \emptyset\rangle$ such that $\mathcal{I}(p) = 1$ if $p = p_1$, 0 otherwise and $\mathcal{O}(p) = 1$ if $p \in \{K, p_1\}$, 0 otherwise; $\beta_2 = \langle \mathcal{I}, \mathcal{O}, \emptyset\rangle$ such that $\mathcal{I}(p) = 1$ if $p = p_1$, 0 otherwise and $\mathcal{O}(p) = 1$ if $p = l_1$, 0 otherwise; $\beta_3 = \langle \mathcal{I}, \mathcal{O}, \emptyset\rangle$ such that $\mathcal{I}(p) = 1$ if $p = loc$, 0 otherwise and $\mathcal{O}(p) = 1$ if $p = p_2$, 0 otherwise; $\beta_4 = \langle \mathcal{I}, \mathcal{O}, \emptyset\rangle$ such that $\mathcal{I}(p) = 1$ if $p \in \{K, p_2\}$, 0 otherwise and $\mathcal{O}(p) = 1$ if $p \in p_2$, 0 otherwise. We consider the initial marking $\mathbf{m}$ such that $\mathbf{m}(p_1) = 1$ and for all $p \in \mathcal{P}' \setminus \{p_1\}$, $\mathbf{m}(p) = 0$. Furthermore, we consider the marking $\mathbf{m}_s$ defined from the configuration s as follows: $\mathbf{m}_s(p_1) = 0$, $\mathbf{m}_s(p_2) = 1$, $\mathbf{m}_s(l) = 0$ for any $l \in L$, $\mathbf{m}_s(c_1) = v^1$, $\mathbf{m}_s(c_2) = v^2$, $\mathbf{m}_s(K) = 0$, and $\mathbf{m}_s(T) = 0$. Let us now show that (i) s is reachable in C iff (ii) $\mathbf{m}_s$ is reachable from $\mathbf{m}$ in $\mathcal{N}'$.

$(i) \rightarrow (ii)$. If s is reachable in C, there exists a computation $\gamma = \langle loc_1, v_1^1, v_1^2\rangle, \langle loc_2, v_2^1, v_2^2\rangle, \ldots, \langle loc_r, v_r^1, v_r^2\rangle$ with $s = \langle loc_r, v_r^1, v_r^2\rangle$ hence $loc = loc_r$, $v^1 = v_r^1$ and $v^2 = v_r^2$. Let us denote by k the maximum of $c_1 + c_2$ along γ. Let us show that we can fire the sequence of multi-transfers $\sigma = \beta_1^k \beta_2 \alpha(\gamma) \beta_3 \beta_4^{k - v_r^1 - v_r^2}$ and that $\mathbf{m} \rightarrowtail^{\sigma} \mathbf{m}_s$. By firing $\beta_1^k \beta_2$, we put k tokens in the capacity place K and one token in the control place l_1. The widget, following Lemma 1, is now ready to simulate faithfully γ by firing the sequence of multi-transfers $\alpha(\gamma)$ as K contains enough tokens. As the simulation was faithful, at the end the place c_1 contains v_r^1 tokens and the place c_2 contains v_r^2 tokens. We also know that the place T contains no tokens, and so by

Lemma 4 the place K contains $k - v_r^1 - v_r^2$ tokens. After, we can fire β_3. The control token is moved from the control location loc_r of the widget to the place p_2. So firing $\beta_4^{k-v_r^1-v_r^2}$ leads to the marking $\mathbf{m}_s$.

$(ii) \to (i)$. Let us make the hypothesis that $\mathbf{m}_s$ is reachable in $\mathcal{N}'$ with a sequence of transitions σ from $\mathbf{m}$. Let us show that σ must be of the form $\beta_1^*\beta_2\sigma_0\beta_3\beta_4^*$, where σ_0 are transitions of the widget. In $\mathbf{m}$, β_1 and β_2 are the only firable transitions. Once β_2 is fired, place l_1 is marked and the transitions σ_0 of the widget have to be fired. To put one token in p_2, transition β_3 has to be fired. After firing β_3, β_4 is the only firable transition. It remains us to prove that $\alpha^{-1}(\sigma_0)$ is a computation of the two counter machine C that reaches s. As $\mathbf{m}_s(T)$ contains no token, by Lemma 2, we know that the simulation was faithful and so $\alpha^{-1}(\sigma)$ leads to s in C. $\Box$

Theorem 3 The action-based LTL model checking problem is undecidable for labeled PN +NBA.

Proof. Let $C = \langle c_1, c_2, L, \mathsf{Instr}\rangle$ be a 2CM and let $s = \langle loc, v^1, v^2\rangle$ be a configuration of C. Let us show that we can reduce the (undecidable) reachability problem of s in C to the action-based LTL model checking problem for a labelled PN + NBA.

We construct a PN + NBA $\mathcal{N}' = \langle \mathcal{P}', \mathcal{B}'\rangle$ starting from the simulation widget $\mathcal{N}_C = \langle \mathcal{P}, \mathcal{B}\rangle$ associated to C. To the simulation widget, we add places and multi-transfers as indicated in Figure 2.4(c). That is, $\mathcal{P}' = \mathcal{P} \cup \{p_1, p_2, p_3\}$, $\mathcal{B}' = \mathcal{B} \cup \{\beta_1, \beta_2, \beta_3, \beta_4, \beta_5\}$ and the new multi-transfers are defined as follows: $\beta_1 = \langle \mathcal{I}, \mathcal{O}, \emptyset\rangle$ such that $\mathcal{I}(p) = 1$ if $p = p_1$, 0 otherwise and $\mathcal{O}(p) = 1$ if $p \in \{K, p_1\}$; $\beta_2 = \langle \mathcal{I}, \mathcal{O}, \emptyset\rangle$ such that $\mathcal{I}(p) = 1$ if $p = p_1$, 0 otherwise and $\mathcal{O}(p) = 1$ if $p = l_1$, 0 otherwise; $\beta_3 = \langle \mathcal{I}, \mathcal{O}, \emptyset\rangle$ such that $\mathcal{I}(p) = v_1$ if $p = c_1$, $\mathcal{I}(p) = v_2$ if $p = c_2$, $\mathcal{I}(p) = 1$ if $p = loc$, 0 otherwise and $\mathcal{O}(p) = 1$ if $p = p_2$, 0 otherwise; $\beta_4 = \langle \mathcal{I}, \mathcal{O}, \{\langle\langle\{c_2\}, T\rangle, 1\rangle\}\rangle$ such that $\mathcal{I}(p) = 1$ if $p = p_2$, 0 otherwise and $\mathcal{O}(p) = 1$ if $p = p_3$, 0 otherwise; $\beta_5 = \langle \mathcal{I}, \mathcal{O}, \{\langle\langle\{c_1\}, T\rangle, 1\rangle\}\rangle$ such that $\mathcal{I}(p) = 1$ if $p = p_3$, 0 otherwise and $\mathcal{O}(p) = 1$ if $p = l_1$, $\mathcal{O}(p) = v_1 + v_2$ if $p = K$, 0 otherwise. The labeling function $\mathcal{L}$ is the identity function, that is for any $M \in \mathcal{B}'$ we have $\mathcal{L}(M) = M$. We consider the initial marking $\mathbf{m}$ such that $\mathbf{m}(p_1) = 1$ and for all $p \in \mathcal{P}' \setminus \{p_1\}$, $\mathbf{m}(p) = 0$. Furthermore, we consider the marking $\mathbf{m}_s$ defined from the configuration s as follows: $\mathbf{m}_s(p_1) = 0$, $\mathbf{m}_s(p_2) = 0$, $\mathbf{m}_s(p_3) = 0$, $\mathbf{m}_s(loc) = 1$, $\mathbf{m}_s(l) = 0$ for any $l \in L \setminus \{loc\}$, $\mathbf{m}_s(c_1) = v^1$, $\mathbf{m}_s(c_2) = v^2$, $\mathbf{m}_s(K) = 0$, and $\mathbf{m}_s(T) = 0$. Let us now show that (i) s is reachable in C iff (ii) $\mathcal{L}(\mathcal{N}', \mathbf{m}) \not\models \neg\Box\Diamond\beta_3$.

$(i) \to (ii)$. If s is reachable in C, there exists a computation $\gamma = \langle loc_1, v_1^1, v_1^2\rangle, \langle loc_2, v_2^1, v_2^2\rangle, \ldots, \langle loc_r, v_r^1, v_r^2\rangle$ with $s = \langle loc_r, v_r^1, v_r^2\rangle$. Let us note k the maximum of $c_1 + c_2$ along γ. We now construct from γ a computation σ of $\mathcal{N}'$ such that $\sigma \models \Box\Diamond\beta_3$. We extend the markings $\mathbf{m}_k$ $(k \geq 0)$ to $\mathcal{P}'$ such that $\mathbf{m}_k(\{p_1, p_2, p_3\}) = 0$. The sequence of multi-transfers $\alpha(\gamma)$ is such that $\mathbf{m} \rightarrowtail^{\beta_1^k\beta_2} \mathbf{m}_k \rightarrowtail^{\alpha(\gamma)} \mathbf{m}_s \rightarrowtail^{\beta_3\beta_4\beta_5} \mathbf{m}_k$. By firing $\beta_1^k\beta_2$, we put k tokens in the capacity place K and one token in the control place l_1 to reach the marking $\mathbf{m}_k$. The widget, following Lemma 1, is now ready to simulate faithfully γ leading to $\mathbf{m}_s$ by firing the sequence of multi-transfers $\alpha(\gamma)$ as K contains enough tokens. After firing β_3, the control token is moved from the control location loc of the widget to the place p_2, v^1 tokens are removed from c_1 and v_2 tokens are removed from c_2. Firing $\beta_4\beta_5$ moves the control token from p_2 to l_1 passing through p_3 and puts $v_1 + v_2$ into K. Since the counters are empty before firing $\beta_4\beta_5$, the transfers of β_4 and β_5 have no effect and we reach $\mathbf{m}_k$. We conclude that the infinite sequence of multi-transfers $\sigma = \beta_1^k\beta_2(\alpha(\gamma)\beta_3\beta_4\beta_5)^\omega$ is firable from $\mathbf{m}$ and satisfies the formula $\Box\Diamond\beta_3$.

$(ii) \to (i)$. Let us make the hypothesis that there is a computation of $\mathcal{N}'$ from the marking $\mathbf{m}$ and satisfying the formula $\Box\Diamond\beta_3$. Let us show that the infinite sequence of multi-transfers σ corresponding to such a computation must be of the form $\beta_1^*\beta_2\sigma_0\beta_3\beta_4\beta_5\ldots\sigma_n\beta_3\beta_4\beta_5\ldots$, where $\sigma_i(i \geq 0)$ is a sequence of multi-transfers of the widget. In fact, β_1 and β_2 are the only firable multi-transfers from $\mathbf{m}$. Once β_2 is fired, the place l_1 is marked and a sequence of multi-transfers of the widget σ_0 must be fired. After firing β_3, β_4 followed by β_5 are the only firable multi-transfers, then a sequence of multi-transfers of the widget σ_1 must be fired, etc.

Suppose that s is not reachable and let us derive a contradiction. Assume that we have $\mathbf{m}_0 \rightarrowtail^{\sigma_0} \ldots \rightarrowtail^{\beta_3\beta_4\beta_5} \mathbf{m}_{2i} \rightarrowtail^{\sigma_i} \mathbf{n}_{2i} \rightarrowtail^{\beta_3\beta_4\beta_5} \mathbf{m}_{2i+1} \rightarrowtail^{\sigma_{i+1}} \ldots$ For each $i \geq 0$, two cases are possible:

1. $\mathbf{m}_{2i}(c_1) = \mathbf{m}_{2i}(c_2) = 0$. We consider here two sub-cases.

   - $(1a)$ $\mathbf{n}_{2i}(c_1) = v^1$ and $\mathbf{n}_{2i}(c_2) = v^2$. As we suppose that s is not reachable, we have that $\alpha^{-1}(\sigma_i)$ does not correspond to a computation of C and, by lemma 3, we know that at least one token has been added to the place T. By lemma 5, one token has been lost from the set of places $\{c_1, c_2, K\}$. Moreover, since $\mathbf{n}_{2i}(c_1) = v^1$ and $\mathbf{n}_{2i}(c_2) = v^2$, all the tokens removed from c_1 and c_2 are put into K and no tokens are removed from K when firing $\beta_3\beta_4\beta_5$ from $\mathbf{n}_{2i}$. So we can conclude that $\mathbf{m}_{2i+1}(\{c_1, c_2, K\}) < \mathbf{m}_{2i}(\{c_1, c_2, K\})$.
   - $(1b)$ $\mathbf{n}_{2i}(c_1) > v^1$ and $\mathbf{n}_{2i}(c_2) \geq v^2$, or $\mathbf{n}_{2i}(c_1) \geq v^1$ and $\mathbf{n}_{2i}(c_2) > v^2$. As no multi-transfer of the widget removes tokens from T, and as $\mathbf{m}_{2i}(\{c_1, c_2, K, T\}) = \mathbf{n}_{2i}(\{c_1, c_2, K, T\})$ by lemma 5, we have that $\mathbf{m}_{2i}(\{c_1, c_2, K\}) \geq \mathbf{n}_{2i}(\{c_1, c_2, K\})$. After firing the sequence $\beta_3\beta_4\beta_5$, at least one token was added to T from the places c_1 or c_2 (through the transfer part of β_4 and β_5). By definition of the multi-transfers β_3, β_4 and β_5, we have $\mathbf{n}_{2i}(\{c_1, c_2, K, T\}) = \mathbf{m}_{2i+1}(\{c_1, c_2, K, T\})$ and we conclude that $\mathbf{m}_{2i+1}(\{c_1, c_2, K\}) < \mathbf{n}_{2i}(\{c_1, c_2, K\})$. Finally, we obtain $\mathbf{m}_{2i+1}(\{c_1, c_2, K\}) < \mathbf{m}_{2i}(\{c_1, c_2, K\})$.

   So in the two sub-cases, we conclude that $\mathbf{m}_{2i+1}(\{c_1, c_2, K\}) < \mathbf{m}_{2i}(\{c_1, c_2, K\})$.

2. $\mathbf{m}_{2i}(c_1) \neq 0$ or $\mathbf{m}_{2i}(c_2) \neq 0$. In that case, we start from a marking $\mathbf{m}_{2i}$ that does not correspond to an initial configuration of the 2CM. Looking at the proof of Lemma 4, we see that it is not possible to add tokens in the set of places $\{c_1, c_2, K\}$ from $\mathbf{m}_{2i}$ to $\mathbf{m}_{2i+1}$ (in fact, we can only move some tokens from $\{c_1, c_2, K\}$ to T). After firing σ_i, two cases are possible.

   - $(2a)$ $\mathbf{n}_{2i}(c_1) = v^1$ and $\mathbf{n}_{2i}(c_2) = v^2$. In that case, firing $\beta_3\beta_4\beta_5$, we reach a marking $\mathbf{m}_{2i+1}$ to which we can apply case 1 above.
   - $(2b)$ $\mathbf{n}_{2i}(c_1) > v^1$ and $\mathbf{n}_{2i}(c_2) \geq v^2$, or $\mathbf{n}_{2i}(c_1) \geq v^1$ and $\mathbf{n}_{2i}(c_2) > v^2$. In that case, after firing the sequence $\beta_3\beta_4\beta_5$, at least one token was added to T from the places c_1 or c_2. Since $\mathbf{m}_{2i}(\{c_1, c_2, K, T\}) = \mathbf{n}_{2i}(\{c_1, c_2, K, T\})$ by lemma 5,we have $\mathbf{m}_{2i+1}(\{c_1, c_2, K\}) < \mathbf{m}_{2i}(\{c_1, c_2, K\})$.

From cases 1 and 2 above, we have that if s is not reachable in C, at least one token is lost in $\{c_1, c_2, K\}$ (at least one token is put in T) when firing $\sigma_i\beta_3\beta_4\beta_5\sigma_{i+1}\beta_3\beta_4\beta_5$ for any $i \geq 0$. This guarantees that the number of tokens in $\{c_1, c_2, K\}$ will become negative after a finite amount of time, which is impossible. We conclude that σ cannot be infinite and, then, cannot satisfy the formula $\Box\Diamond\beta_3$. This contradicts our hypothesis. $\Box$

Notice that the state-based LTL model checking is undecidable for Petri nets, so do for MTNs.

Theorem 4 The place boundedness problem is undecidable for $\mathsf{PN}+\mathsf{NBA}$.

Proof. Let $C = \langle c_1, c_2, L, \mathsf{Instr}\rangle$ be a 2CM and let $s = \langle loc, v^1, v^2\rangle$ be a configuration of C. Let us show that we can reduce the (undecidable) boundedness problem of C to the place boundedness problem for a $\mathsf{PN}+\mathsf{NBA}$.

We construct a $\mathsf{PN}+\mathsf{NBA}$ $\mathcal{N}' = \langle \mathcal{P}', \mathcal{B}'\rangle$ starting from the simulation widget $\mathcal{N}_C = \langle \mathcal{P}, \mathcal{B}\rangle$ (schematized in Figure 2.4(a)) associated to C. To the simulation widget, we add places and multi-transfers as indicated in Figure 2.4(d). That is, $\mathcal{P}' = \mathcal{P} \cup \{p_1, p_2\}$, $\mathcal{B}' = \mathcal{B} \cup \{\beta_1, \beta_2, \beta_3, \beta_4\} \cup \{\beta_{l_i} | \mathsf{TypeInst}(l_i) = \mathsf{inc}_j\}$ and the new multi-transfers are defined as follows: $\beta_1 = \langle \mathcal{I}, \mathcal{O}, \emptyset\rangle$ such that $\mathcal{I}(p) = 1$ if $p \in \{p_1, c_1\}$, 0 otherwise and $\mathcal{O}(p) = 1$ if $p \in \{p_1, K\}$, 0 otherwise; $\beta_2 = \langle \mathcal{I}, \mathcal{O}, \emptyset\rangle$ such that $\mathcal{I}(p) = 1$ if $p \in \{p_1, c_2\}$, 0 otherwise and $\mathcal{O}(p) = 1$ if $p \in \{p_1, K\}$, 0 otherwise; $\beta_3 = \langle \mathcal{I}, \mathcal{O}, \{\langle\langle\{c_1\}, T\rangle, 1\rangle\}\rangle$ such that $\mathcal{I}(p) = 1$ if $p = p_1$, 0 otherwise and $\mathcal{O}(p) = 1$ if $p = p_2$, 0 otherwise; $\beta_4 = \langle \mathcal{I}, \mathcal{O}, \{\langle\langle\{c_2\}, T\rangle, 1\rangle\}\rangle$ such that $\mathcal{I}(p) = 1$ if $p = p_2$, 0 otherwise and $\mathcal{O}(p) = 1$ if $p = l_1$, 0 otherwise; $\beta_{l_i} = \langle \mathcal{I}, \mathcal{O}, \{\langle\langle\{K\}, T\rangle, 1\rangle\}\rangle$ such that $\mathcal{I}(p) = 1$ if $p = l_i$, 0 otherwise and $\mathcal{O}(p) = 1$ if $p = p_1$, 0 otherwise. We extend the markings $\mathbf{m}_k$ $(k \geq 1)$ to $\mathcal{P}'$ such that $\mathbf{m}_k(\{p_1, p_2\}) = 0$ and we consider $\mathbf{m}_0$ as the initial marking. Let us now show that (i) C is unbounded iff (ii) K is unbounded in $\mathcal{N}'$.

$(i) \to (ii)$ If C is unbounded, then (since only one counter may be increased by 1 at a time) there exists an infinite set of computations of C $\{\gamma_0, \gamma_1, \ldots\}$ such that we have $\gamma_i = \langle loc_{i1}, v^1_{i1}, v^2_{i1}\rangle \ldots \langle loc_{ik_i}, v^1_{ik_i}, v^2_{ik_i}\rangle$ with $v^1_{il} + v^1_{il} \leq i$ for any $1 \leq l \leq k_i$, $v^1_{ik_i} + v^2_{ik_i} = i$ and $\mathsf{TypeInst}(loc_{ik_i}) = inc_j$ [4]. The sequences of multi-transfers $\alpha(\gamma_i)(i \geq 0)$ are such that $\mathbf{m}_0 \rightarrowtail^{\alpha(\gamma_0)\beta_{l_{i_0}}\beta_3\beta_4} \mathbf{m}_1 \rightarrowtail^{\alpha(\gamma_1)\beta_{l_{i_1}}(\beta_1|\beta_2)\beta_3\beta_4} \ldots \mathbf{m}_k \rightarrowtail^{\alpha(\gamma_k)\beta_{l_{i_k}}(\beta_1|\beta_2)^k\beta_3\beta_4} \mathbf{m}_{k+1} \ldots$ The widget, following Lemma 1, is ready to simulate faithfully γ_k from any $\mathbf{m}_k$ $(k \geq 1)$ leading to a marking where K contains no token and from which some multi-transfer $\beta_{l_{i_k}}$ is firable. After firing $\beta_{l_{i_k}}$, the control token is moved to the place p_1. The k tokens in $\{c_1, c_2\}$ are moved to K by firing β_1 or β_2 k times. Then, firing $\beta_3\beta_4$ moves the control token from p_1 to l_1 passing through p_2 and a new token is generated into K. Moreover, since c_1 and c_2 are empty before firing $\beta_3\beta_4$, the transfers of β_3 and β_4 have no effect (no tokens are transferred from c_1 or c_2 to T). We conclude that we have $\mathbf{m}_0 \rightarrowtail^{\alpha(\gamma_0)\beta_{l_{i_0}}\beta_3\beta_4} \mathbf{m}_1 \rightarrowtail^{\alpha(\gamma_1)\beta_{l_{i_1}}(\beta_1|\beta_2)^1\beta_3\beta_4} \ldots \mathbf{m}_k \rightarrowtail^{\alpha(\gamma_k)\beta_{l_{i_k}}(\beta_1|\beta_2)^k\beta_3\beta_4} \mathbf{m}_{k+1} \ldots$ in $\mathcal{N}'$ where K is unbounded.

$(ii) \to (i)$ Let us make the hypothesis that the place K is unbounded. Let us show that for any $c \in \mathbb{N}$, a sequence of multi-transfers that allows us to reach from $\mathbf{m}_0$ a marking $\mathbf{m}$ with $\mathbf{m}(K) = c$ has the form $\sigma_0\beta_{l_{i_0}}(\beta_1|\beta_2)^*\beta_3\beta_4 \ldots \sigma_j\beta_{l_{i_j}}(\beta_1|\beta_2)^*\beta_3\beta_4 \ldots \sigma_n\beta_{l_{i_n}}(\beta_1|\beta_2)^*\beta_3\beta_4$ where each σ_i $(0 \leq i \leq n)$ is a sequence of multi-transfers of the widget. β_4 is the only multi-transfer that allows the net to increase the number of tokens in $\{c_1, c_2, K\}$ and it adds exactly one token into this set of places. From $\mathbf{m}_0$, multi-transfers of the widget can be fired until a multi-transfer $\beta_{l_{i_1}}$ is fired. Firing $\beta_{l_{i_1}}$ moves the control token to p_1. While the control token stays into the place p_1, β_1 and β_2 can be fired and move one token from c_1 to K and from c_2 to K, respectively. Then, $\beta_3\beta_4$ can be fired moving the control token from p_1 to l_1 through p_2 and one token is generated into K. If there are some tokens into c_1 and c_2 before

[4] Notice that since C is deterministic, we have that γ_i is a prefix of γ_{i+1} $(i \geq 0)$.

firing $\beta_3\beta_4$, at least one token is moved to T. This allows the net to reach a marking from which the widget is ready to simulate the 2CM and a similar reasoning can be applied, etc.

Let us prove that from a marking $\mathbf{m}$ with $\mathbf{m}(\{c_1, c_2, K\}) = k$, we can reach a marking $\mathbf{m}'$ with $\mathbf{m}'(\{c_1, c_2, K\}) = k+1$ by firing a sequence of multi-transfers σ of the widget and then a sequence of multi-transfers of the form $\beta_l(\beta_1|\beta_2)^*\beta_3\beta_4$ only if before firing β_l the place K is empty and before firing $\beta_3\beta_4$ the multi-transfers β_1 and β_2 are fired until all the tokens in c_1 and c_2 have been moved into K. So to increase the number of tokens in $\{c_1, c_2, K\}$, the places c_1 and c_2 must be emptied. Indeed, suppose it is not the case. From Lemma 5 and the fact that the tokens in T cannot move anymore, there should exist markings $\mathbf{n}, \mathbf{n}'$ and $\mathbf{n}''$ and a multi-transfer $\beta_{l_{i_l}}$ such that $\mathbf{n} \rightarrowtail^{\beta_{l_{i_l}}(\beta_1|\beta_2)^*} \mathbf{n}' \rightarrowtail^{\beta_3\beta_4} \mathbf{n}''$ with either $\mathbf{n}(K) \neq 0$ or $\mathbf{n}'(\{c_1, c_2\}) > 0$, and $\mathbf{n}(\{c_1, c_2, K\}) < \mathbf{n}''(\{c_1, c_2, K\})$. In the case where $\mathbf{n}(K) \neq 0$, we have $\mathbf{n}'(\{c_1, c_2, K\}) < \mathbf{n}(\{c_1, c_2, K\})$ as the transfer of $\beta_{l_{i_l}}$ transfers one token from K to T and no tokens are added into $\{c_1, c_2, K\}$ when firing $\beta_{l_{i_l}}$, β_1 and β_2. In this case, more than one token is added into $\{c_1, c_2, K\}$ when firing $\beta_3\beta_4$ as we have $\mathbf{n}(\{c_1, c_2, K\}) < \mathbf{n}''(\{c_1, c_2, K\})$. This contradicts the fact that β_4 is the only multi-transfer that adds tokens into $\{c_1, c_2, K\}$ and it adds exactly one token into this set of places.

In the case where we have $\mathbf{n}'(\{c_1, c_2\}) > 0$, then $\beta_3\beta_4$ moves at least one token from $\{c_1, c_2\}$ to T with the bounded transfers of β_3 and β_4. We conclude that we have $\mathbf{n}''(\{c_1, c_2, K\}) \leq \mathbf{n}'(\{c_1, c_2, K\})$ as β_3 does not add tokens in $\{c_1, c_2, K\}$ and β_4 adds exactly one token into this set of places. We conclude that there exist sequences of multi-transfers of the form $\beta_{l_{i_l}}(\beta_1|\beta_2)^*$ that generate tokens in $\{c_1, c_2, K\}$. This makes a contradiction with the fact that β_4 is the only multi-transfer that adds tokens in this set of places.

Let us assume that the 2CM C is bounded by c and that we have $\mathbf{m}_0 \rightarrowtail^{\sigma_0\beta_{l_{i_0}}(\beta_1|\beta_2)^*\beta_3\beta_4} \ldots \mathbf{m}_j \rightarrowtail^{\sigma_j\beta_{l_{i_j}}(\beta_1|\beta_2)^*\beta_3\beta_4} \ldots \mathbf{m}_n \ldots$ with $\mathbf{m}_j(K) = c+1$, $\mathbf{m}_j(\{c_1, c_2\}) = 0$ and $\mathbf{m}_j(l_1) = 1$.This marking $\mathbf{m}_j$ corresponds to a marking reached after incrementing the number of tokens in $\{c_1, c_2, K\}$ to obtain $c+1$ tokens in this set, i.e. $\mathbf{m}_j(\{c_1, c_2, K\}) = \mathbf{m}_{j-1}(\{c_1, c_2, K\}) + 1$. Indeed, as proved just before, the places c_1 and c_2 must be emptied to increase the number of tokens in $\{c_1, c_2, K\}$. So to have $c+1$ tokens in $\{c_1, c_2, K\}$, we need to reach such a marking at some point of the run of $\mathcal{N}'$. The following cases hold.

1. $\alpha^{-1}(\sigma_j)$ is a computation of C. In this case, following Lemma 3 and Lemma 5, the marking reached from $\mathbf{m}_j$ by firing σ_j has at least one token in K and the transfer of $\beta_{l_{i_j}}$ moves one token from K to T when $\beta_{l_{i_j}}$ is fired. Moreover, the following sub-cases holds.

   - We have $\mathbf{m}_j \rightarrowtail^{\sigma_j\beta_{l_{i_j}}(\beta_1|\beta_2)^*} \mathbf{m} \rightarrowtail^{\beta_3\beta_4} \mathbf{m}_{j+1}$ with $\mathbf{m}(\{c_1, c_2\}) > 0$. In that case, the transfers of β_3 and β_4 move at least one token from $\{c_1, c_2, K\}$ to T and we have $\mathbf{m}_{j+1}(\{c_1, c_2, K\}) < c$. As proved just before, to have again $c+1$ tokens in $\{c_1, c_2, K\}$, $\mathcal{N}'_C$ will have to reach again a marking $\mathbf{m}_n$ where there is no tokens in $\{c_1, c_2\}$ and $c+1$ tokens in K.

   - We have $\mathbf{m}_j \rightarrowtail^{\sigma_j\beta_{l_{i_j}}(\beta_1|\beta_2)^*} \mathbf{m} \rightarrowtail^{\beta_3\beta_4} \mathbf{m}_{j+1}$ with $\mathbf{m}(\{c_1, c_2\}) = 0$. In that case, we have $\mathbf{m}_{j+1}(K) = c+1$ and $\mathbf{m}_{j+1}(\{c_1, c_2\}) = 0$. We conclude that we can have the same reasoning from $\mathbf{m}_{j+1}$ than from $\mathbf{m}_j$.

2. $\alpha^{-1}(\sigma_j)$ is not a computation of C. Following Lemma 3, one token moves from K to T when σ_j is fired. Moreover, the following sub-cases holds.

- We have $\mathbf{m}_j \rightarrowtail^{\sigma_j \beta_{l_{i_j}} (\beta_1|\beta_2)^*} \mathbf{m} \rightarrowtail^{\beta_3\beta_4} \mathbf{m}_{j+1}$ with $\mathbf{m}(\{c_1, c_2\}) > 0$. In that case, the transfers of β_3 and β_4 move at least one token from $\{c_1, c_2, K\}$ to T and we have $\mathbf{m}_{j+1}(\{c_1, c_2, K\}) < c$. As proved just before, to have again $c+1$ tokens in $\{c_1, c_2, K\}$, $\mathcal{N}'_C$ will have to reach again a marking $\mathbf{m}_n$ where there is no tokens in $\{c_1, c_2\}$ and $c+1$ tokens in K.
- We have $\mathbf{m}_j \rightarrowtail^{\sigma_j \beta_{l_{i_j}} (\beta_1|\beta_2)^*} \mathbf{m} \rightarrowtail^{\beta_3\beta_4} \mathbf{m}_{j+1}$ with $\mathbf{m}(\{c_1, c_2\}) = 0$. In that case, we have $\mathbf{m}_{j+1}(K) = c+1$ and $\mathbf{m}_{j+1}(\{c_1, c_2\}) = 0$. We conclude that we can have the same reasoning from $\mathbf{m}_{j+1}$ than from $\mathbf{m}_j$.

We conclude that if $\mathcal{N}'_C$ can reach a marking where there are $c+1$ tokens in $\{c_1, c_2, K\}$, then it must reach a marking $\mathbf{m}$ where c_1 and c_2 are empty, K contains $c+1$ tokens and from which it can start to simulate C. Moreover, from this marking, $\mathcal{N}'_C$ reaches after simulating C and firing $(\beta_1 \mid \beta_2)^*\beta_3\beta_4$ either a marking similar to $\mathbf{m}$ in the sense that c_1 and c_2 are empty, K contains $c+1$ tokens and $\mathcal{N}'_C$ can start to simulate C again, or it reaches a marking where there are less that $c+1$ tokens in $\{c_1, c_2, K\}$ and $\mathcal{N}'_C$ can start to simulate C again. In this last case, to reach again a marking with $c+1$ tokens in $\{c_1, c_2, K\}$, $\mathcal{N}'_C$ must reach again a marking where there is no tokens in c_1 and c_2, $c+1$ tokens in K and from which $\mathcal{N}'_C$ can start to simulate C, i.e. a marking similar to $\mathbf{m}$. We conclude that $\mathcal{N}'_C$ can never reach a marking where there are $c+2$ tokens in K, hence $\mathcal{N}'_C$ cannot have more than $c+1$ tokens in the set of places $\{c_1, c_2, K\}$, i.e.K is bounded. $\square$

Conclusion

Although Petri nets with transfers and Petri nets with non-blocking arcs model different kinds of systems, decidability and undecidability results are the same for the two models and for the considered problems. This can be viewed as *positive*, as some properties can be automatically checked on PN + NBA, but also as *negative*, as a lot of decidable problems on Petri nets become undecidable on PN + NBA. Table 2.2 reports the results obtained in this part.

Model	Coverability	Boundedness	Place boundedness	LTL model checking	Reachability
PN	D	D	D	D	D
PN + T	D	D	I	I	I
MTN	D	D	I	I	I
PN + NBA	D	D	I	I	I

Table 2.2: Decidability and undecidability results on (extended) Petri nets.

2.3 From Global Machines to MTNs

Following [BCR01, Del00, DB01], most of the properties that we want to check on a parameterized system do not depend on the identity of the different identical components. Such properties can be decided for global machines on an abstract model where the different instances of local machines cannot be distinguished i.e. where each instance of a local machine

$\mathcal{L}_i$ is mapped to a black token moving in the places of a MTN corresponding to the current local states of $\mathcal{L}_i$. This approach is particularly interesting for parameterized verification (see [GS92, EFM99]).

The counting abstraction that allows us to reason about global machines using MTNs is defined as follows.

Definition 15 (Counting Abstraction) Let $\mathcal{G} = \langle \mathbf{B}, \langle \mathcal{L}_1, k_1 \rangle, \ldots, \langle \mathcal{L}_m, k_m \rangle \rangle$ be a global machine in which $\mathcal{L}_i$ has n_i local states, i.e., $Q_i = \{s_{i1}, \ldots, s_{in_i}\}$. Given a global state $G = \langle \rho, \mathbf{s} \rangle$, we define α as:

$$\alpha(G) = \langle \rho, \langle v_{11}, \ldots, v_{1n_1}, \ldots, v_{m1}, \ldots, v_{mn_m} \rangle \rangle,$$

where v_{ij}=number of *occurrences* of the state $s_{ij} \in Q_i$ in $\mathbf{s}$.

Thus, α maps a global state in a tuple whose length is always $size(G) = |\mathbf{B}| + n_1 + \ldots + n_m$ for any value of $k_1, \ldots, k_m$ (i.e. independently from the number of processes). Furthermore, it maps all symmetric states into a unique representation.

For instance, for the global machine $\langle \{b\}, \langle \mathcal{L}_1, 3 \rangle \rangle$, the two states $\langle \langle true \rangle, \langle wait, wait, use \rangle \rangle$ and $\langle \langle true \rangle, \langle wait, use, wait \rangle \rangle$ are mapped into the abstract state $\langle \langle true \rangle, \langle 2, 1, 0, \ldots, 0 \rangle \rangle$, where the first component of the tuple $\langle 2, 1, \ldots \rangle$ denotes the number of *wait*, and so on.

Given a collection of local machines, we can *automatically* extract an abstract system represented as an MTN as we will describe in the following.

Formal Definition of the Translation

Let $G = \langle \mathbf{B}, \langle \mathcal{L}_1, k_1 \rangle, \ldots, \langle \mathcal{L}_m, k_m \rangle \rangle$ be a *global machine*, with $\mathcal{L}_i = \langle Q_i, \Sigma_i, \delta_i \rangle$. Then, we define the corresponding MTN $\mathcal{M}_G = \langle \mathcal{P}, \mathcal{B} \rangle$ as the MTN in which the set of places $\mathcal{P}$ is defined as $Q_1 \cup \ldots \cup Q_m \cup \{true_b, false_b | b \in \mathbf{B}\}$ (if we assume that $Q_i \cap \{true_b, false_b | b \in \mathbf{B}\} = \emptyset$ for all $1 \leq i \leq m$, i.e. no local state has the name of a place used to encode a boolean variable). Intuitively, the number of tokens in the place q with $q \in Q_i$ $(1 \leq i \leq m)$ will define the number of local machine in the local state q and each pair of place $\{true_b, false_b\}$ will always contain exactly one token. This token in $true_b$ ($false_b$) will mean that the variable b is equal to true (false).

The translation of the transitions in G works by grouping together sending and receptions associated to a single label a.

Specifically, let us suppose that in G there is exactly one pair of rendez-vous send and reception for each label α (the generalization corresponds to consider all the possible pairs of rendez-vous send and reception and make the treatment presented here for each of those pairs). Then, for *each transition* $s \xrightarrow{\alpha:\varphi} s'$of some local machine, such that α is either an *internal move*, *a rendez-vous send*, a *signal send* or a *broadcast send*, we define a multi-transfer $M_{\alpha:\varphi} = \langle T, S_1, S_\infty \rangle$. The transition $T = \langle \mathcal{I}, \mathcal{O} \rangle$, and the transfer blocks $S_1 = \langle \{\ldots\}, 1 \rangle$, and $S_\infty = \langle \{\ldots\}, \infty \rangle$ are defined *incrementally* according to the following rules. We set initially $S_1 = \langle \emptyset, 1 \rangle$, $S_\infty = \langle \emptyset, \infty \rangle$ and $\mathcal{I}(p) = \mathcal{O}(p) = 0$ for any $p \in P$.

Boolean Guard/Actions If b occurs in a literal ℓ of φ, and $b' = c$ occurs in φ, we model the effect of φ on b as two Petri net edges in T that move one token from the place naturally associated to the usage of b to the place associated to c. E.g. for $\neg b \wedge b' = true$ we need a transition that tests and removes one token from $false_b$ and adds it to $true_b$. If b does not occur as a guard in φ, then we add a transfer in S_∞ to transfer all the tokens from one of the two places $true_b$ and $false_b$ to the other. E.g. for $b' = true$ we have a transfer that goes from $false_b$ to $true_b$. Formally,

- $\mathcal{I}(true_b) = 1$ if φ contains a test on b;
- $\mathcal{I}(false_b) = 1$ if φ contains a test on $\neg b$;
- $\mathcal{O}(true_b) = 1$ if φ contains a test on b and assigns $true$ to b/leaves it unchanged, or φ contains a test on $\neg b$ and sets it to $true$;
- $\mathcal{O}(false_b) = 1$ if φ contains a test on $\neg b$ and assigns $false$ to b/leaves it unchanged, or φ contains a test on b and sets it to $false$;
- finally, if φ does not contain a test on b or $\neg b$ but it contains an assignment on b', then we add the transfer $B_b = \langle P_b, p_b \rangle$ to S_∞, where:
  - if φ contains $b' = false$, then $P_b = \{true_b\}$ and $p_b = false_b$;
  - if φ contains $b' = true$, then $P_b = \{false_b\}$ and $p_b = true_b$.

For each variable b such that φ does not contain a test on b or $\neg b$ and $b' = b$ occurs in φ we need no actions: by default the value of places remained unchanged when firing a MTN multi-transfer.

Internal action An internal action can be modeled, by adding an extra input edge (from the source state) and an extra output edge (to the target state) to the Petri net transition part T. Specifically, if α is an internal move from s to s', then we set $\mathcal{I}(s) = 1$ and $\mathcal{O}(s') = 1$.

Rendez-vous The translation is similar to internal moves, but we We need here two edges (from the source and to target states) for the sender and two edges for the receiver. Specifically, suppose that $\alpha = a!$ and that the transition goes from s to s', and that reception with label $a?$ goes from t to t'. Then, we set $\mathcal{I}(s) = 1$ and $\mathcal{O}(s') = 1$, and $\mathcal{I}(t) = 1$ and $\mathcal{O}(t') = 1$.

Signal Signal can be modeled by adding a transfer to S_1 (at most 1 process receives the message!). The send part of the signal (the edge labeled with $a \uparrow$) is modeled with edges in the Petri net transition part T and transfers are used to model the receive part (edges labeled by $a \downarrow$). Each transfer models the sources and target places of the set of receivers that move to the same state. Specifically, let $\alpha = a\uparrow$ and suppose that the transition goes from s to s'. We first set $\mathcal{I}(s) = 1$ and $\mathcal{O}(s') = 1$. Then, for each $p \in Q_i$ for some $1 \leq i \leq m$ such that there exists p' with $p' \xrightarrow{a\downarrow} p \in \delta_i$ we add to S_1 the transfer $B = \langle P, p \rangle$ where $P = \{p' \mid p' \xrightarrow{a\downarrow} p \in \delta_i\}$.

Broadcast A broadcast can be naturally modeled via transfers in S_∞. Specifically, an input (from the source state of the sender) and an output (to the target state of the sender) edge are added to the Petri net transition part T. Finally, a transfer for each set of receivers with the same target state is added to the set of transfers of S_∞. Specifically, let $\alpha = a!!$ and suppose the transition goes from s to s'.We first set $\mathcal{I}(s) = 1$ and $\mathcal{O}(s') = 1$. Then, for each $p \in Q_i$ for some $1 \leq i \leq m$ such that there exists p' with $p' \xrightarrow{a??} p \in \delta_i$ we add to S_∞ the transfer $B = \langle P, p \rangle$ where $P = \{p' \mid p' \xrightarrow{a??} p \in \delta_i\}$.

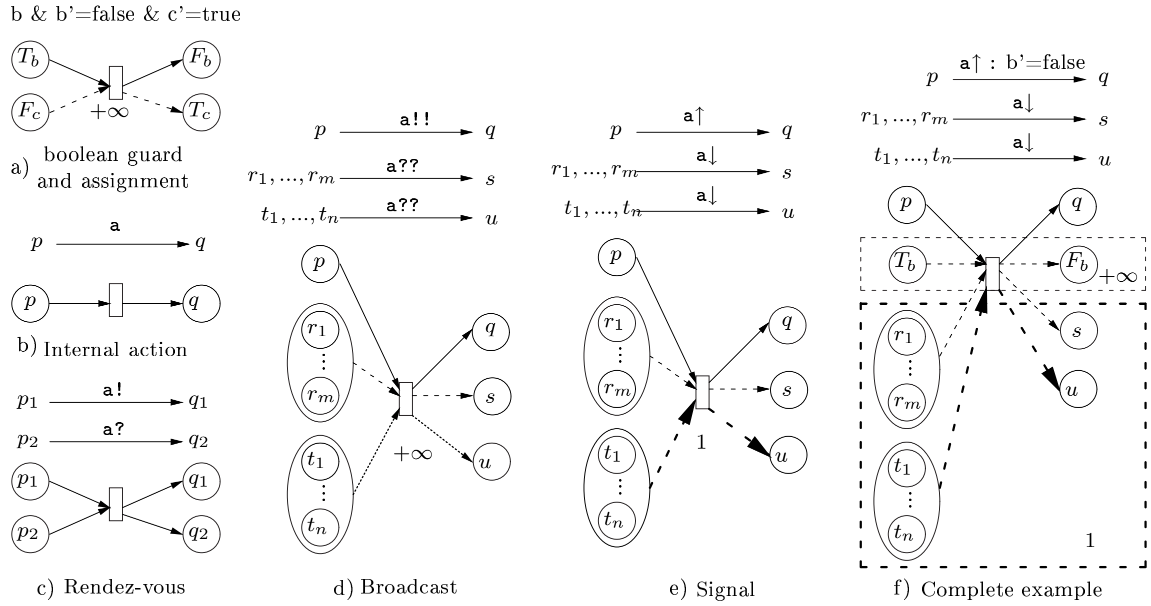

Figure 2.5: From global machines rules to MTNs.

Fig. 2.5 summarizes the translation from a global machine to a single MTN.

Notice that $\mathcal{M}_\mathcal{G}$ is a MTN, i.e. each the multi-transfer M of $\mathcal{M}_\mathcal{G}$ with the set of transfer blocks $\{S_1, \ldots, S_u\}$ satisfies the following conditions:

1. for any *transfer block* S_k with the set of *transfers* $\{B_{k1}, \ldots, B_{kr_k}\}$, and for any such B_{kj}, we require that $p_{kj} \notin P_{kj}$;

2. for any *transfer* $B_{ki} = \langle P_{ki}, p_{ki} \rangle$ in the transfer block S_k and $B_{lj} = \langle P_{lj}, p_{lj} \rangle$ in the transfer block S_l with $B_{ki} \neq B_{lj}$, we require that $(P_{ki} \cup \{p_{ki}\}) \cap (P_{lj} \cup \{p_{lj}\}) = \emptyset$.

Indeed, the transfers describing assignments to boolean variables satisfy the two conditions by construction. For the transfers that describes the reception of a message, the first condition is ensured since the machines that receive a broadcast message or a signal must react by changing their current local state (Condition 1 of Definition 3).

The second condition is ensured by the fact that (i) a local state in which the local machine can receive a signal $a\downarrow$ (a broadcast a??) cannot be the target local state in which the local machine moves by receiving $a\downarrow$ (a broadcast a??) (Condition 1 of Definition 3), and (ii) a local machine moves to a unique local state (there is no non-determinism) when receiving a broadcast message or a signal (see Condition 2 of Definition 3) (and also the fact that the sets of local states of local machines are disjoint).

Based on the previous ideas, we have the following proposition.

Proposition 2 Let $\mathcal{G}$ be a global machine and $\mathcal{M}_\mathcal{G}$ be the MTN constructed from the transformation of section 2.3.

Then, the following holds.

$$\mathbf{m}_0\mathbf{m}_1 \ldots \text{ is a run in } \mathcal{M} \text{ iff } G_0G_1 \ldots \text{ is a run in } \mathcal{G}$$

and $\mathbf{m}_i$ uniquely represents $\alpha(G_i)$. Thus, the reachability set of $\mathcal{M}$ is a *conservative approximation* of that of $\mathcal{G}$.

Given a global state G of the global machine $\mathcal{G}$, we denote in the following the marking of $\mathcal{M}_\mathcal{G}$ that uniquely represents $\alpha(G)$ by $\alpha_{\mathcal{M}_\mathcal{G}}(G)$.

Proposition 2 allows us to reduce the local state reachability problem (see Definition 8) for global machines to the marking coverability problem for MTNs. Let φ be a formula of the propositional logic on the global boolean variables of $\mathcal{G}$ and f_S be a function that assigns a natural number to each local state. To decide if there exists a reachable global state where the boolean variables satisfy φ and where for each local states s we have at least $f_S(s)$ local machines in s, we proceed as follows. We construct the set of markings S containing the markings where

1. the tokens in the places $\{true_b, false_b \mid b \text{ is a boolean variable of the global machine }\}$ corresponds to a valuation of the global variables that satisfies φ. More precisely, we have one token into the pairs of places $\{true_b, false_b\}$ and the valuation defined by the configuration of tokens into those places satisfies φ;

2. for each place q corresponding to the local state q, we have $f_S(q)$ tokens in q.

Those markings are the minimal ones that correspond to a global state satisfying the constraints defined by φ and f_S. Assume that G_0 is the initial global state. There is a marking $\mathbf{m}$ that is reachable from $\alpha_{\mathcal{M}_\mathcal{G}}(G_0)$ in $\mathcal{M}_\mathcal{G}$ such that there is $\mathbf{m}' \in S$ with $\mathbf{m}' \preccurlyeq \mathbf{m}$ if and only if there is a global state G that is reachable from G_0 in $\mathcal{G}$ that satisfies the constraints defined by φ and f_S.

More formally, we have the following.

Theorem 5 Let $\mathcal{G} = \langle \mathbf{B}, \mathcal{L}_1, \ldots, \mathcal{L}_k \rangle$ be a global machine and $\mathcal{M}_\mathcal{G}$ be the MTN constructed from $\mathcal{G}$ applying the transformation presented in this section.

Then, the following holds.
$\mathcal{G}$ with the initial global state G_0 satisfies the local state reachability for the formula φ of the propositional logic and the function f_S if and only if $S \cap \mathsf{Reach}_{\mathcal{M}_\mathcal{G}}(\alpha_{\mathcal{M}_\mathcal{G}}(G_0)) \neq \emptyset$ where $S = \{\mathbf{m} \mid \text{there exists a global state } G \text{ such that } G = \langle \Gamma, \Theta \rangle \text{ with } \Gamma(\varphi) \text{ and } \forall s \in Q_1 \cup \ldots \cup Q_k : \alpha_{\mathcal{M}_\mathcal{G}}(G)(s) = f_S(s), \text{ and } \alpha_{\mathcal{M}_\mathcal{G}}(G) \preccurlyeq \mathbf{m}\}$.

Proof. From Proposition 2, we obtain that $\alpha_{\mathcal{M}_\mathcal{G}}(G) \in \mathsf{Reach}_{\mathcal{M}_\mathcal{G}}(\alpha_{\mathcal{M}_\mathcal{G}}(G_0))$ if and only if G is reachable from G_0 in $\mathcal{G}$. The set of markings that are greater or equal to a marking in S contains all the markings corresponding to global states satisfying the constraints induced by φ and f_S plus markings that does not corresponds to global states. Those markings contain more than one token in the pairs of places $\{true_b, false_b\}$ (where b is a global variable of $\mathcal{G}$). The construction of $\mathcal{M}_\mathcal{G}$ ensures that the number of tokens in the pairs of places $\{true_b, false_b\}$ is always constant. We conclude that the markings that are greater or equal to a marking in S that do not correspond to global states are not reachable from G_0. Finally, there exists $G = \langle \Gamma, \Theta \rangle$ that is is reachable from G_0 with $\Gamma(\varphi)$ and $\forall s \in Q_1 \cup \ldots \cup Q_k : \alpha_{\mathcal{M}_\mathcal{G}}(G)(s) \geq f_S(s)$ if and only if there is a marking $\mathbf{m} \in \mathsf{Reach}_{\mathcal{M}_\mathcal{G}}(\alpha_{\mathcal{M}_\mathcal{G}}(G_0))$ and a marking $\mathbf{m}' \in S$ with $\mathbf{m}' \preccurlyeq \mathbf{m}$.
□

Chapter 3

Abstraction of Concurrent Programs

In this chapter, we focus on the construction of formal models describing the behavior of software programs. More precisely, we define a simple core programming language, called *Concurrent Boolean Programs* (CBPs for short), that extends the boolean programs of T. Ball and S.K. Rajamani [BR00]. This language is designed to be close to *real life concurrent programming language* like JAVA [Gra97b, Fla97], c with POSIX threads [But97], Ada [Bar96], etc. As a consequence, concurrent boolean programs can be automatically obtained from real life programs by applying the predicate abstraction [GS97].

We then explain how to encode the behavior of concurrent boolean programs with an (potentially) unbounded number of threads to a family of global machines, i.e. a set of local machines where the number of instantiations can be a parameter.

Finally, we show how JAVA programs can be modeled using concurrent boolean programs and abstraction techniques like the predicate abstraction [GS97].

The content of this chapter is based on the paper [DRVB02].

3.1 Concurrent Boolean Programs

In this section, we introduce the *language of Concurrent Boolean Programs* (CBP for short)[1]. In the subsection 3.1.1, we present the *syntax* of this language as well as a *first definition* of what a CBP means. In the subsection 3.1.2, we present the semantics of the language, thanks to a second, more formal, definition of CBP (the two definitions are shown however to be equivalent). The semantics is stated through a set of *axioms*, in a fashion similar to the one in the discussion of chapter 8 of [AO97].

3.1.1 Syntax

We begin by introducing the syntax of CBP. When designing this language, we have tried to keep as close as possible to the JAVA programing language, but without losing the ability to translate other multi-threaded languages (like Ada or C with POSIX threads) into CBP.

[1]In the sequel, we will use the notation CBP to speak about *the language of CBPs*, as well as a concurrent boolean program – a piece of code written in the language of CBP. The context should be sufficient to let the reader to find out which meaning should be applied...

As said in the introduction of the chapter, CBPs can be regarded as a natural concurrency-aimed extension of the Boolean Programs, defined by T. Ball and S.K. Rajamani (see for instance [BR00]). As the name suggests, a CBP is made of several pieces of code that we will call *threads*, in a fashion similar to Java. These threads are rather independent processing units, that will be able to run in *parallel*, following an interleaving semantics (as we will show in the next section). More precisely, we have extended Ball and Rajamani's boolean programs by means of the following primitives: `sleep`, `wakeup`, `wakeupall`, `start`, `accept` and `rendezvous`. In order to fully capture the synchronisation semantics of Java, which requires the ability to let a thread lock an object, we have also added the concept of *lock*.

It is worth noticing that the *boolean* variables handled by a CBP might be of two types. On the one side, we have the *global variables*, that can be read and written by any thread. On the other side, we have several sets of local variables, which belong to a single thread, and can be accessed by their sole owner.

In this section we give the necessary syntactic rules to write a well-formed CBP. The first (classical) step toward this achievement, is to state the *grammar* of CBP. Such a grammar is shown in Figure 3.1. We have taken the convention to write the *keywords* of the language (as `start` or `sleep`) in `typewriter face` ; and all the other terminals (like the variables names: **varName**) in **bold face**.

Example 1 *An example of a very simple CBP is given in Figure 3.2. It comprises three threads:* `t1`*,* `t2` *and* `main`*. The program begins with a declaration section. We first declare the global variables (in this case:* `g1` *and* `g2`*), the locks (no locks are declared in this example), the messages (a single message* `msg` *is declared here) and the names of the thread types. The definition of each type of thread – which begins by the declaration of the local variables – is then given.*

We can already briefly sketch the semantics of this example: an initial copy of the `main` *thread will* start *one copy of* `t1` *and one copy of* `t2` *(assigning* `false` *to the local variable* `l1` *and* `true` *to* `l2`*). The running instance of* `t2` *will try to synchronise with the instance of* `t1` *through the* `accept` *statement (the receiving side of the rendez-vous). After some tests and assignments, the instance of* `t1` *will send the rendez-vous, along with the value stored in* `l1`*. This value will be assigned to* `l2` *by the instance of* `t2`*.*

In order to simplify and clarify the subsequent discussion, we need to add, to the grammar, a set of more restrictive well-formedness rules. These rules are stated in the next subsection.

Well-Formed Concurrent Boolean Programs

We begin this subsection with a set of useful definitions:

Let $\mathit{Locks}(B)$, $\mathit{Global}(B)$, $\mathit{Messages}(B)$ be respectively the set of locks, global variables, and messages declared in the CBP B. Let $\mathit{NameThreads}(B)$ be the set of names of thread types declared in the preamble of B, and let $\mathit{Threads}(B)$ be the set of thread types whose code is specified in B. Furthermore, let $\mathit{Labels}(T)$ be the set of labels present in the definition of the thread type $T \in \mathit{Threads}(B)$ (that is, for every ℓ in $\mathit{Labels}(T)$, there exists an command in T beginning by $[\ell]$, and conversely). $\mathit{Labels}(B)$ is the set of labels appearing in the definitions of all the thread type $T \in \mathit{Threads}(B)$. Finally, let $\mathit{Local}(T)$ be the set of *local variables* declared for the thread type T, and $\mathit{Body}(T)$ be the sequence of instructions ('LabeledCommands' in Figure 3.1) obtained by derivation of 'ThreadBody', for the thread type T.

[1]	Program	→	Declarations Threads$^+$
[2]	Declarations	→	VarDec LockDec MsgDec ThreadDec
[3]	VarDec	→	`vars :` **varName*** `;`
[4]	LockDec	→	`locks :` **lockId*** `;`
[5]	MsgDec	→	`messages :` **msgId*** `;`
[6]	ThreadDec	→	`threads :` **threadTypeName**$^+$ `;`
[7]	Threads	→	**threadTypeName** `{` VarDec ThreadBody `}`
[8]	ThreadBody	→	LblCommands$^+$
[9]	LblCommands	→	`[`**label**`]` Command
[10]		→	Command
[11]	Command	→	Assignment `;`
[12]		→	`if (`Condition`)` `{`LblCommands$^+$`}` `else` `{`LblCommands$^+$`}`
[13]		→	`choice {` GuardedAssign$^+$ `}`
[14]		→	`while (`Condition`)` `{` LblCommands$^+$ `}`
[15]		→	`skip ;`
[16]		→	`goto (`**label**`) ;`
[17]		→	`lock (`**lockId**`) ;`
[18]		→	`unlock (`**lockId**`) ;`
[19]		→	`sleep (`**msgId**`,` **lockId**`) ;`
[20]		→	`wakeup (`**msgId**`) ;`
[21]		→	`wakeupall (`**msgId**`) ;`
[22]		→	`start (`**threadTypeName**`) ;`
[23]		→	`start (`**threadTypeName**`,` Parameters`) ;`
[24]		→	`rendezvous (`**msgId**`) ;`
[25]		→	`rendezvous (`**msgId**`,` ExpressionList`) ;`
[26]		→	`accept (`**msgId**`) ;`
[27]		→	`accept (`**msgId**`,` VariableList`) ;`
[28]	Condition	→	`*`
[29]		→	Expression
[30]	GuardedAssign	→	Expression `:` Assignment `;`
[31]	Assignment	→	**varName** AssignMiddle Litteral
[32]	AssignMiddle	→	`,` **varName** AssignMiddle Litteral`,`
[33]		→	`:=`
[34]	Parameters	→	Litteral`,` Parameters
[35]		→	Litteral
[36]	ExpressionList	→	Expression `,` ExpressionList
[37]		→	Expression
[38]	VariableList	→	**varName**`,` VariableList
[39]		→	**varName**
[40]	Expression	→	**varName**
[41]		→	`!` **varName**
[42]		→	Expression `and` Expression
[43]		→	Expression `or` Expression
[44]		→	Litteral
[45]	Litteral	→	`true`
[46]		→	`false`

Figure 3.1: Grammar for the Concurrent boolean programs

```
vars :  g1 g2 ;
locks :  ;
messages :  msg ;
threads :  t1 t2 main ;

t1 { vars :  l1 ;
   if (g1) {
      skip ;
   } else {
      l1 := true ;
   }
   rendezvous(msg, l1) ;
}

t2 { vars :  l2 ;
   skip ;
   accept (msg, l2) ;
}

main { vars :  ;
   start(true, false) ;
   start(false, true) ;
}
```

Figure 3.2: A simple Concurrent Boolean Program.

A *well-formed* concurrent boolean program is a program B, generated by the grammar shown at Figure 3.1 which also satisfies the following conditions:

Labels In B, we cannot find two commands labelled with the same label;

Names The sets *Threads*(B), *Locks*(B), *Global*(B), *Messages*(B) and *Labels*(T), *Local*(T) for all $T \in$ *Threads*(B) are mutually exclusive. This means that there is no element of one of those sets that is also an element of another set. Mainly, this rule avoids the case where a variable is defined as global variable and as local variable of a thread and allows us to clarify the presentation;

Goto For each instruction `goto`(ℓ) in the thread body of $T \in$ *Threads*(B): ℓ must belong to *Labels*(T);

Declarations If a variable v is used in the body of the thread type T, v must be either in *Global*(B), or in *Local*(T). Each lock that is used in B (in a `lock`, an `unlock` or a `sleep` instruction) must belong to *Locks*(B). Each thread type in *NameThreads*(B) must also be in *Threads*(B) and conversely. Moreover, for each $T \in$ *Threads*(B), the code of T is defined only once in B. Finally, each message used in B (in a `sleep`, a `wakeup(all)`, a `rendezvous` or an `accept`) must be part of *Messages*(B);

Thread start For each `start`$(T, \pi_1, \ldots, \pi_n)$ instruction in B, two conditions must hold: 1) $T \in$ *Threads*(B) and 2) the number n of parameters and the number of local variables of thread type T must be equal. If we have a command `start`(T) in B, then *Local*$(T) = \emptyset$;

Main thread There must be a `main` thread declared in B: $\{$`main`$\} \in$ *NameThreads*(B);

Rendez-vous For each `rendezvous`$(M, e_1, \ldots, e_n)$ in the body of a thread type T, every e_i $(1 \leq i \leq n)$ must be an expression over variables in *Local*(T). We impose this restriction to simplify the presentation but it is not limitative. Indeed, to take into account the value of a global variable, a thread can copy its value into one of its local variables before executing the `rendezvous`;

Accept For each $\texttt{accept}(M, u_1, \ldots, u_n)$ appearing in the body of a thread type T, all the u_i ($1 \leq i \leq n$) must be in $\mathit{Local}(T)$. As in the previous case, the restriction is to simplify the presentation but it is not limitative. Indeed, if a thread wants to put a value received using an `accept` operation into a global variable, it can receive it and put it into a local variable and then copy the value from its local variable to the global variable.

Moreover, we impose that for all $1 \leq i \neq j \leq n$ we have $u_i \neq u_j$;

Assignment For each $u_1, \ldots, u_n{:=}v_1, \ldots, v_n$ in the body of any thread, we have for all $1 \leq i, j \leq n$ that $i \neq j \Rightarrow u_i \neq u_j$;

Locks The `lock` and `unlock` operations must always be paired at the same `if` or `while` block level. By this constraint we mean that the instructions of the form $\texttt{lock}(L)$ and $\texttt{unlock}(L)$ are coupled in sequences of instructions derived from LblCommands in the grammar and of the form $B_1 \cdot \texttt{lock}(L)\ ; \cdot B_2 \cdot \texttt{unlock}(L)\ ; \cdot B_3$ (we use $\cdot$ as a *sequence operator*). Moreover, we impose that B_2 does not contain any other instructions $\texttt{lock}(L)$ and $\texttt{unlock}(L)$, i.e. there is no imbrication of pairs of instructions $\texttt{lock}(L)$ and $\texttt{unlock}(L)$ for a same lock L.

For instance, we cannot have something like

$$\texttt{lock}(L)\ ;\texttt{if}(E)\ \texttt{then}\{\texttt{unlock}(L)\ ;\}\texttt{else}\{\texttt{skip}\ ;\}$$

Moreover if a `goto` appears inside a `lock`/`unlock` block, then it must branch to a label *inside* the block. Symmetrically, no `goto` outside of a `lock`/`unlock` block can branch to a label inside the block. Of course, the `lock` must happen *before* the corresponding `unlock`.

We impose these conditions to simplify the presentation and the extension to the general case where the operations `lock` and `unlock` can appear in the boolean programs without restrictions can be achieved by keeping track of the locks taken by threads in their local state. More precisely, this rule has been introduced to avoid situations in which a thread unlocks a lock that it doesn't hold and the situation where a thread tries to take a lock that it already has. In the particular case we are interested in (the translation of a JAVA program into a CBP), this restriction is not annoying, since JAVA owns a special `synchronized` statement that allows us to build a complete block protected by a single lock;

Sleep For the same reasons than in the previous condition, we impose that an instruction of the form $\texttt{sleep}(M, L)$; is always in a block B_2 in a block of instructions B of the form $B_1 \cdot \texttt{lock}(L)\ ; \cdot B_2 \cdot \texttt{unlock}(L)\ ; \cdot B_3$.

Definition 16 ((Well-Formed) Concurrent Boolean Program – First definition) A *(Well-Formed) Concurrent Boolean Program* is any program written in the programming language defined with the grammar[2] of Figure 3.1, and following the well-formedness rules of 3.1.1.

[2] We assume here that the rules of the grammar are not ordered. We consider that the priority of boolean operators in boolean expressions is the following: ! has the highest priority, then `or`, and finally `and`.

3.1.2 Semantics

We now address the behavioral aspect of CBP. We achieve it in this section by presenting a set of *axioms* that allows us to describe the *operational semantics* of CBP. These axioms are mainly inspired by the discussion of Chapter 8 of [AO97].

We open this section with some useful definitions that state precisely what we mean by 'Concurrent Boolean Program'. We also define the concept of *state* of a CBP, which is the basis to a formal operational semantics. Finally, we present the axioms and we give the definition of an *execution of a CBP*.

Definitions

Up to now, we have seen a CBP as a set of *types of thread*, plus some *global variables* and *locks* (recall the declaration section of 3.1.1). We can now state this more precisely.

First, a *type of thread* contains all the necessary information to *instantiate* a new thread.

Definition 17 (Type of Thread) A *Type of Thread* is a tuple $\langle \mathcal{V}, S \rangle$, where:

- $\mathcal{V}$ is a set of *local boolean variables* $\{v_1, \ldots, v_n\}$. In the following, we assume a total order on these variables (corresponding to the order of declaration in the program built from the grammar of section 3.1.1): this order is important for the `start` operation;

- S is a sequence $I_1 \cdot I_2 \cdot \ldots \cdot I_n$ of instructions. Each such instruction can be derived from the LabeledCommands rule of the grammar at Figure 3.1 and the sequence S satisfies the well-formedness rules.

Definition 18 (Concurrent Boolean Program (CBP)) A **Concurrent Boolean Program** is a tuple $\langle \mathcal{G}, \mathcal{C}, \mathcal{T} \rangle$, where:

- $\mathcal{G}$ is a set of *global boolean variables* $\{v_1, \ldots, v_m\}$;

- $\mathcal{C}$ is a set of *locks* $\{l_1, \ldots, l_k\}$;

- $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ is a set of *types of thread* where $T_i = \langle \mathcal{V}_i, S_i \rangle$ and $\mathcal{V}_i \cap \mathcal{G} = \emptyset$ for all $1 \leq i \leq k$. We denote the singleton $\{T_j\}$ as T_j.

This definition of a CBP is completely equivalent to Definition 16. Indeed, given a CBP B_1 constructed thanks to the rules of 3.1.1, we can construct an equivalent CBP $B_2 = \langle \mathcal{G}, \mathcal{C}, \mathcal{T} \rangle$. More precisely, B_2 has the following properties.

1. For each variable in $Global(B_1)$, there exists one and only one variable in $\mathcal{G}$;

2. For each lock in $Locks(B_1)$, there exists one and only one lock in $\mathcal{C}$;

3. For each thread T in $Threads(B_1)$, there exists one and only one type of thread $\langle \mathcal{V}, S \rangle$ in $\mathcal{T}$, such that:

    (a) For each variable in $Local(T)$, there exists one and only one variable in $\mathcal{V}$;

    (b) S is exactly $Body(T)$.

In the same way, it is easy to be convinced that from any CBP $B_2 = \langle \mathcal{G}, \mathcal{C}, \mathcal{T} \rangle$ we can construct an equivalent well-formed CBP B_1 constructed thanks to the rules of 3.1.1 (if we assume a renaming of the local variables to ensure $\mathcal{V}_i \cap \mathcal{V}_j = \emptyset$ for all $i \neq j$).

Example 2 *For instance, the type of thread for the thread* `t2` *of Figure 3.2 is*

$$\langle \underbrace{\{\texttt{l2}\}}_{\textit{local variables}}, \underbrace{\texttt{skip} \;;\cdot\, \texttt{accept(msg,l2)} \;;}_{\textit{sequence of instructions}} \rangle$$

It might sound quite cumbersome to have two different definitions for the same concept. However, syntactical matters about CBPs are more easily introduced through the grammar of Figure 3.1, hence from Definition 16. On the other hand, the semantical aspect of the CBPs can be very easily introduced in a formal program-rewriting system. Therefore, we need to have a definition such as Definition 18 at our disposal.

With this new definition, we can now explain the behavior of a CBP. In its initial state, a special thread (called `main`) is the only one to be active (such a thread always exists: see the well-formedness rules of 3.1.1). Each thread has the possibility to start several other threads of a given type, thanks to the `start` primitive. It should be clear that the evolution of a CBP being in a given 'state' depends only on the evolution of the different thread instances that are active in that state: we do not need more information in order to be able to predict the future evolution of the CBP.

Therefore, we define the notions of *global state* of a CBP, which depends on the *local states* of the thread instances, and on the valuations of the global variables and on the locks. The idea of the local state of thread instance is captured through the *thread instantiation* definition, which gives the valuation of the local variables of the thread, as well as the point of the code that has been reached so far.

Definition 19 (Global State of a Concurrent Boolean Program.) A **Global State of a CBP** $\langle \mathcal{G}, \mathcal{C}, \mathcal{T} \rangle$ is a tuple $\langle \Gamma, \Lambda, \{\Theta\} \rangle$, where:

- $\Gamma : \mathcal{G} \mapsto \{true, false\}$ is a valuation of the global variables. We extend this valuation to expressions over boolean variables (see rule 40 of Figure 3.1) as follows:
  - $\Gamma(\texttt{true}) = true$
  - $\Gamma(\texttt{false}) = false$
  - $\Gamma(!u) = true$ if $\Gamma(u) = false$, $false$ otherwise
  - $\Gamma(u_1 \texttt{ and } u_2) = true$ if $\Gamma(u_1) = true$ and $\Gamma(u_2) = true$, $false$ otherwise
  - $\Gamma(u_1 \texttt{ or } u_2) = true$ if $\Gamma(u_1) = true$ or $\Gamma(u_2) = true$, $false$ otherwise
- $\Lambda : \mathcal{C} \mapsto \{\textit{locked}, \textit{unlocked}\}$ is a valuation of the locks;
- Θ is a multi-set of thread instantiations of types in $\mathcal{T}$.

As we will see in the remaining of this section, threads can fall into *wait* states by executing the `sleep` operation. Intuitively, the `sleep` operation unlocks the lock given in parameter and then the thread falls into a waiting state. Threads in such a state are blocked until they synchronize with another thread that executes a `wakeup` or a `wakeupall` operation. To model a waiting state, we need to extend the grammar of CBP with a special *wait* command. We define the *extended CBP grammar* as the grammar shown in Figure 3.1 extended with the following rule:

$$\text{Command} \rightarrow \textit{wait}(\textbf{msgIdM},\textbf{lockId});$$

A thread instantiation is defined as follows.

Definition 20 (Thread Instance) A *thread instance* of $T_i = \langle \mathcal{V}_i, S_i \rangle$ $(1 \leq i \leq k)$ is a tuple $\langle P, \rho \rangle$, such that

- $P = I_1 \cdot \ldots \cdot I_n$ denotes the sequence of commands remaining to be executed where each I_i can be derived from the LlbCommand rule of the extended CBP grammar ($P = \varepsilon$ if no instruction remains). P is not guaranteed to satisfies all the well-formed rules (for instance, a label could appear several times in the following);

- $\rho : \mathcal{V}_i \mapsto \{true, false\}$ is a valuation of the local variables. ρ is extended to the expressions over boolean variables similarly to Γ (see Definition 19).

In the following, we will sometime use the term *local state* instead of thread instance.

Let $B = \langle \mathcal{G}, \mathcal{C}, \mathcal{T} \rangle$ be a CBP where the type `main` of threads is $\langle \mathcal{V}, S \rangle$. The initial global state of S is the tuple $init_B = \langle \Gamma_I, \Lambda_I, \{I\} \rangle$ where

- for all $v \in \mathcal{G}$ we have $\Gamma(v) = true$;

- for all $l \in \mathcal{C}$ we have $\Lambda(l) = \textit{unlocked}$;

- $I = \langle S, \rho_I \rangle$ where for all $v \in \mathcal{V}$ we have $\rho_I(v) = true$.

Axioms

The axioms given in this subsection allow us to compute all the legal transitions between two global states of a CBP. Two types of operations executed by threads can be distinguished:

1. **The local operations.** These operations are executed by one thread and have no effect on the other ones.

   Among these axioms, one of them is of particular interest: the axiom 3.6 for the `goto`, because of the definition of function γ we have to introduce there (it will be re-used in the sequel). Roughly speaking, this function allows us to know, given a program S and a label ℓ, what are the instructions of S that remain to be executed from the operation labelled by ℓ. γ is necessary to handle correctly the cases where, e.g., we make a jump to a label that is into several nested `while` or `if` blocks. In these cases, it should be obvious that the instructions to execute after the one at label ℓ might not be these that immediately follow ℓ in S.

2. **The synchronisation operations.** These operations are executed by a thread but modify the local state of other ones.

Along this section, we assume a mapping from the set of tokens **lockId**, **varName** and **threadTypeName** to their corresponding instances in the sets $\mathcal{C}$, $\mathcal{G} \cup_i \mathcal{V}_i$ and $\mathcal{T}$, respectively. We use M, L and T to denote a message, a lock and a type of thread respectively. Moreover, given a valuation ρ over a set of variables V, we will sometimes denote by $\rho[x_1 = c_1 \ldots x_n = c_n]$ the valuation ρ' such that for all $x \in V$, we have $\rho'(x) = c_i$ if $x = x_i$ $(1 \leq i \leq n)$, $\rho'(x) = \rho(x)$ otherwise. We will also make no distinction between the literals `true` and `false` used in the grammar and their semantics (in the assignment and the choice rule).

The axioms for `rendezvous`(m), `accept`(m), `start`(T) and the labelled version of commands have been omitted. For `rendezvous` and `accept` the axioms can be deduced from the axioms 3.14 (for `accept` and `rendezvous`) and 3.13 (for `start`), with an empty set of parameters. For the labelled version of the commands, the rules are exactly the same than the rules for the non-labelled version of the commands.

Assignment Rule

$$\begin{array}{c}\langle \Gamma, \Lambda, \{\langle v_1, \ldots, v_n := u_1, \ldots, u_n; \cdot S' \ , \rho\rangle\} + \Sigma\rangle \\ \rightarrow \\ \langle \Gamma[v_{i_1} = u_{i_1} \ldots v_{i_k} = u_{i_k}], \Lambda, \{\langle S', \rho[v_{i_{k+1}} = u_{i_{k+1}} \ldots v_{i_n} = u_{i_n}]\rangle\} + \Sigma\rangle\end{array} \tag{3.1}$$

where the set of local variables of the thread is $\mathcal{V}$ and Σ is a multi set of thread instances.We also assume a permutation of the variable indexes: $\{1, \ldots, n\} \rightarrow \{i_1, \ldots, i_n\}$, in order to *sort the variables by type*. Indeed, our permutation is such that: $\forall j : \ 1 \leq j \leq k : \ v_{i_j} \in \mathcal{G}$ and $\forall j : k < j \leq n : v_{i_j} \in \mathcal{V}$ (k may be 0 or n).

Assignments in CBP can only be used to set the value of variables to true or false (see Figure 3.1). However, it is possible to assign values depending on the values of variables (for instance assign the value of the variable y to the variable x) using the `choice` command (see Rule 3.3).

If Rule

$$\begin{array}{c}\langle \Gamma, \Lambda, \{\langle \texttt{if}\ (E)\ \texttt{then}\ \{S_1\}\ \texttt{else}\ \{S_2\} \cdot S', \rho\rangle\} + \Sigma\rangle \\ \rightarrow \\ \left\{\begin{array}{ll} \langle \Gamma, \Lambda, \{\langle S_1 \cdot S', \rho\rangle\} + \Sigma\rangle & \text{if } E \equiv * \text{ or if } \rho \cup \Gamma(E) = true \\ \langle \Gamma, \Lambda, \{\langle S_2 \cdot S', \rho\rangle\} + \Sigma\rangle & \text{if } E \equiv * \text{ or if } \rho \cup \Gamma(E) = false \end{array}\right.\end{array} \tag{3.2}$$

Choice rule For every j such that $1 \leq j \leq m$, if $(\Gamma \cup \rho)(E_j)$ we have

$$\begin{array}{c}\langle \Gamma, \Lambda, \{\langle \texttt{choice}\{E_1{:}v_{1,1}, \ldots, v_{1,n_1}{:=}u_{1,1}, \ldots, u_{1,n_1};\ \ldots \\ E_m{:}v_{m,1}, \ldots, v_{m,n_m}{:=}u_{m,1}, \ldots, u_{m,n_m};\} \cdot S', \rho\rangle\} + \Sigma\rangle \\ \rightarrow \\ \langle \Gamma[v_{j,i_1} = u_{j,i_1}, \ldots, v_{j,i_k} = u_{j,i_k}], \Lambda, \{\langle S', \rho[v_{j,i_{k+1}} = u_{j,i_{k+1}}, \ldots, v_{j,i_{m_j}} = u_{j,i_{m_j}}]\rangle\} + \Sigma\rangle\end{array} \tag{3.3}$$

As in the case of the assignment rule we assume a re-ordering of the assignments $v_{j,i} = u_{j,i}$, such that the k first assignments are assignments to global variables, and the remainders are assignments to local variables.

The `choice` command can be used to assign complex values to variables. For instance, given two variables x and y, if we want to assign the value of y to x, it is not possible to do it using a simple assignment. Instead, we use the following command:

```
choice {
y : x:= true;
¬y : x:= false;
}
```

The first assignment in the `choice` can be applied only if y is equal to true and it puts x to true, and the second one can only be applied when y is equal to false and it puts x to false. Hence, this command assigns the value of y into x.

This principle also holds when we want to assign the value of an expression into variables. So, even if assignments are very simple in CBP, they are not limitative.

While Rule

$$\begin{array}{c}\langle \Gamma, \Lambda, \{\langle \texttt{while}\ (E)\{S\} \cdot S', \rho\rangle\} + \Sigma\rangle \\ \rightarrow \\ \left\{\begin{array}{ll}\langle \Gamma, \Lambda, \{\langle S \cdot \texttt{while}\ (E)\{S\} \cdot S', \rho\rangle\} + \Sigma\rangle & \text{if } E \equiv * \text{ or if } \rho \cup \Gamma(E) = true \\ \langle \Gamma, \Lambda, \{\langle S', \rho\rangle\} + \Sigma\rangle & \text{if } E \equiv * \text{ or if } \rho \cup \Gamma(E) = false\end{array}\right.\end{array} \tag{3.4}$$

Skip Rule

$$\langle \Gamma, \Lambda, \{\langle \texttt{skip;} \cdot S'\ , \rho\rangle\} + \Sigma\rangle \rightarrow \langle \Gamma, \Lambda, \{\langle S', \rho\rangle\} + \Sigma\rangle \tag{3.5}$$

Goto Rule

$$\langle \Gamma, \Lambda, \{\langle \texttt{goto}\ (\ell); \cdot S'\ , \rho\rangle\} + \Sigma\rangle \rightarrow \langle \Gamma, \Lambda, \{\langle \gamma(S, \ell), \rho\rangle\} + \Sigma\rangle \tag{3.6}$$

Where $\langle \texttt{goto}\ (\ell); \cdot S', \rho\rangle$ is a thread instance of the thread type $T = \langle \mathcal{V}, S\rangle$ and $\gamma(S, \ell)$ is defined as follows.

Definition 21 (The γ function) Let B be a well-formed CBP and $T \in \mathit{Thread}(B)$ with $T = \langle \mathcal{V}, S\rangle$. Given a label ℓ of T, $\gamma(S, \ell)$ is defined as follows:

1. If $S \equiv S_1 \cdot [\ell]I \cdot S_2$, where S_1 does not contain ℓ, then: $\gamma(S_1, \ell) = [\ell]I \cdot S_2$.
2. If $S \equiv S_1 \cdot [\ell']\ \texttt{while}(E)\{S_2\} \cdot S_3$, where S_1 does not contain ℓ, but S_2 does, then $\gamma(S, \ell) = \gamma(S_2, \ell) \cdot [\ell']\ \texttt{while}(E)\{S_2\} \cdot S_3$;
3. If $S \equiv S_1 \cdot [\ell']\ \texttt{if}(E)\ \{S_2\}\texttt{else}\{S_3\} \cdot S_4$, where S_1 does not contain ℓ and:
   (a) either S_2 contains ℓ, then $\gamma(S, \ell) = \gamma(S_2, \ell) \cdot S_4$;
   (b) or S_3 contains ℓ, then $\gamma(S, \ell) = \gamma(S_3, \ell) \cdot S_4$.

Notice that starting from S and applying γ, the recursive calls of the γ function are always applied on *blocks of commands* of S. As S contains one command labelled by ℓ (following the well-formedness rules), so are the blocks of commands on which the recursive calls are applied. It follows that γ returns exactly one sequence of commands when applied on S and ℓ and, so, is indeed a function.

This γ function will be crucial in the sequel (see Section 3.2). It simply returns, given a thread type $T = \langle \mathcal{V}, S\rangle$ and a label ℓ, the sequence of instructions remaining to be executed when the execution of an instance of T reaches ℓ. For instance, if an instance of the thread type T executes S, which contains ℓ, then the local state of this thread will be exactly $\langle \gamma(S, \ell), \rho\rangle$ for some ρ when the flow of execution of the instance will reach the instruction labelled by ℓ.

Notice that the S' part is lost here : goto is not compositional.

Lock Rule The semantics of the `lock` instruction is simply to *take a lock*. This instruction, along with the corresponding `unlock`, has been introduced in order to model the `synchronized` blocks of JAVA.

$$\langle \Gamma, \Lambda, \{\langle \texttt{lock}\ (L); \cdot S'\ , \rho\rangle\} + \Sigma\rangle \rightarrow \langle \Gamma, \Lambda[L = \mathit{locked}], \{\langle S', \rho\rangle\} + \Sigma\rangle \tag{3.7}$$

iff $\Lambda(L) = \mathit{unlocked}$.

Unlock Rule

$$\langle \Gamma, \Lambda, \{\langle \texttt{unlock}\ (L); \cdot S'\ , \rho\rangle\} + \Sigma\rangle \rightarrow \langle \Gamma, \Lambda[L = \mathit{unlocked}], \{\langle S', \rho\rangle\} + \Sigma\rangle \tag{3.8}$$

Notice that following the well-formedness rules, a thread can only release a lock that it owns, i.e. $\Lambda(L) = \mathit{locked}$.

Sleep Rule The `sleep` instruction has been introduced to model the `wait` instruction of JAVA. The semantics of the latter construct is to let the thread release the last taken lock and jump to a kind of *sleep state*, waiting to be awoken. Therefore, the `sleep` instruction has two parameters: the first one corresponds to the message that will awake the thread ; and the second one is the lock to be relinquished.

To model the *sleep state* of the thread, we use the special instruction *wait* of the extended grammar of CBP labelled by $\mathbf{wait}_\ell$. Notice that for any CBP B, we assume that $\mathbf{wait}_\ell \notin \mathit{Labels}(B)$. ℓ is the label of the `sleep` command that allows the thread to fall into the sleep state. In the following, the label ℓ is used to distinguish the wait states accessed by executing different `sleep` commands. To model that a thread moves to the sleep state when it calls `sleep`, we substitute it by *wait* in the program of the thread. Remark that, after being woken up, the thread immediately tries to re-acquire the lock it had relinquished during its last execution of the `sleep` command (see axioms 3.10 to 3.12).

$$\begin{gathered}\langle \Gamma, \Lambda, \{\langle [\ell]\texttt{sleep}\ (M, L); \cdot S', \rho\rangle\} + \Sigma\rangle \\ \rightarrow \\ \langle \Gamma, \Lambda[L = \mathit{unlocked}], \{\langle [\mathbf{wait}_\ell]\mathit{wait}(M, L); \cdot S', \rho\rangle\} + \Sigma\rangle\end{gathered} \tag{3.9}$$

Notice that following the well-formedness rules, a thread can only release a lock that it owns, i.e. $\Lambda(L) = \mathit{locked}$.

Wakeup Rules The `wakeup` statement has been introduced to model the `notify` statement of JAVA (about the semantics of this JAVA construct, see Section 1.3.1). As a `notify` can have no effect on the other threads if none of them is waiting to be awoken, two rules are necessary for the `wakeup` statement. The first one gives the semantics when there is at least one thread waiting to synchronise on a given message. The second is for the case where no such thread exists. In the following, we consider that $\mathbf{lock}_\ell \notin \mathit{Labels}(B)$ for any CBP B.

$$\begin{gathered}\langle \Gamma, \Lambda, \{\langle \texttt{wakeup}\ (M); \cdot S_1, \rho_1\rangle, \langle [\mathbf{wait}_\ell]\mathit{wait}(M, L); \cdot S_2, \rho_2\rangle\} + \Sigma\rangle \\ \rightarrow \\ \langle \Gamma, \Lambda, \{\langle S_1, \rho_1\rangle, \langle [\mathbf{lock}_\ell]\texttt{lock}\ (L); \cdot S_2, \rho_2\rangle\} + \Sigma\rangle\end{gathered} \tag{3.10}$$

Notice that the starting global state can contain several instances of threads of the form $\langle [\mathbf{wait}_\ell]\mathit{wait}(M, L); \cdot S_2, \rho_2\rangle$. In that case, the local machine used to apply the rule is chosen non-deterministically among the possible local machines.

The *wait* command is replaced by a `lock` command in the second local state that has a role in the rule meaning that the awoken thread will first acquire the lock that it relinquished when it felt asleep before execution the remaining of its code.

$$\langle \Gamma, \Lambda, \{\langle \texttt{wakeup}\ (M); \cdot S, \rho \rangle\} + \Sigma \rangle \rightarrow \langle \Gamma, \Lambda, \{\langle S, \rho \rangle\} + \Sigma \rangle \tag{3.11}$$

iff:

$$\Sigma = \{\langle S_1, \rho_1 \rangle, \langle S_2, \rho_2 \rangle, \ldots, \langle S_i, \rho_i \rangle, \ldots, \langle S_k, \rho_k \rangle\}$$

and

$$\forall j : 1 \leq j \leq k : \forall L \in \mathcal{C} : S_j \not\equiv [\mathbf{wait}_\ell]\mathit{wait}(M, L); \cdot S'_j$$

Wakeupall Rule As we could expect, the `wakeupall` statement has been introduced to model the effect of a `notifyAll` in JAVA. As for the `wakeup`, a `wakeupall` could be of no effect if no threads are waiting to be awoken.

$$\langle \Gamma, \Lambda, \{\langle \texttt{wakeupall}\ (M) \cdot S, \rho \rangle\} + \Sigma_1 + \Sigma_2 \rangle \rightarrow \langle \Gamma, \Lambda, \{\langle S, \rho \rangle\} + \Sigma'_1 + \Sigma_2 \rangle \tag{3.12}$$

where

- for all $I \in \Sigma_1$, we have that I is of the form $\langle [\mathbf{wait}_\ell]\mathit{wait}(M, L) \cdot S_I, \rho_I \rangle$ for some L;
- for all $I \in \Sigma_2$, we have that I is not of the form $\langle [\mathbf{wait}_\ell]\mathit{wait}(M, L) \cdot S_I, \rho_I \rangle$ for some L;
- for all lock L, Σ'_1 contains k copies of the local state $\langle [\mathbf{lock}_\ell]\texttt{lock}\ (L) \cdot S', \rho' \rangle$ if and only if Σ_1 contains k copies of the local state $\langle [\mathbf{wait}_\ell]\mathit{wait}(M, L) \cdot S', \rho' \rangle$.

Start Rule This statement is useful to model the call to the `run` method of a `Thread` object in JAVA. Its parameters are: the name of the type of thread whom we want a new instance to be started, and a set of values, which should be assigned to the local variables of the newly created instance.

$$\langle \Gamma, \Lambda, \{\langle \texttt{start}\ (T_i, \pi_1, \ldots, \pi_{|\mathcal{V}_i|}); \cdot S', \rho \rangle\} + \Sigma \rangle \rightarrow \langle \Gamma, \Lambda, \{\langle S', \rho \rangle, \langle S_i, \rho' \rangle\} + \Sigma \rangle \tag{3.13}$$

where $T_i = \langle \mathcal{V}_i, S_i \rangle \in \mathcal{T}$ and ρ' is such that $\forall i : 1 \leq i \leq |\mathcal{V}_i| : \rho'(v_i) = \pi_i$ (recall that we assume a total order on local variables, corresponding to the order of definition in the program constructed from the grammar of section 3.1.1).

Rendez-vous **Rule** This is the classical blocking synchronisation primitive. A `rendezvous` always has to be paired with a corresponding (w.r.t. the message) `accept`. Our version of the *rendez-vous* also allows us to pass values from the 'sending' thread (calling `rendezvous`) to the 'accepting' thread (doing the `accept`). We have chosen to restrict ourselves to *local values*[3], as a global variable can always be consulted and modified by any thread.

There is no *rendez-vous*-like primitive in JAVA, but we have chosen to retain it in order to model other concurrent languages such as ADA [Bar96]. In ADA, we can define *entry points* in the tasks (the ADA equivalent to threads). These entry points can have parameters, through

[3]Values stored in a local variable, or literals.

which we can pass values to the task implementing them (an entry point in Ada is thus similar to an `accept` in CBP). The '*rendez-vous*' in ADA is implemented as a function call, with the name of the entry points being the name of the function.

$$\begin{array}{c}\langle \Gamma, \Lambda, \{\langle \texttt{accept}\ (M, v_1, \ldots, v_n) \cdot S, \rho\rangle, \langle \texttt{rendezvous}\ (M, u_1, \ldots, u_n) \cdot S', \rho'\rangle\} + \Sigma\rangle \\ \rightarrow \\ \langle \Gamma, \Lambda, \{\langle S, \rho[v_1 = \rho'(u_1), \ldots, v_n = \rho'(u_n)]\rangle, \langle S', \rho'\rangle\} + \Sigma\rangle \end{array} \tag{3.14}$$

Execution of a CBP

The last step to fully achieve the semantics of CBP is to state what is an execution of a CBP.

Definition 22 (Execution of a CBP) Given a CBP B, an *execution* of B is a sequence $e = G_1, G_2, \ldots, G_n$ of global states such that: $\forall i:\ 1 \leq i \leq n-1:\ G_i \rightarrow G_{i+1}$ (according to the previous axioms). G_1 is said to be the *initial state* of e. G_n is the *target state* of e.

The definition of (finite) execution of a CBP can be extended to infinite execution in the obvious way but in the following we will always consider finite ones.

From the definition of execution, we can infer the notion of reachable state:

Definition 23 (Reachable State) Given a CBP B, a global state G_t of B is said to be *reachable* from the global state G_i of B (written $G_i \Rightarrow G_t$) iff there exists an execution e of B such that G_i is the initial state of e, and G_t is the target state of e.

In the next chapters, we will focus on the verification of CBPs. More precisely, we will focus on the verification of a particular class of reachability properties called *control states reachability properties*. Those problems consist to decide if there is a reachable global state of the CBP where the global variables satisfy a property (given by a propositional formula over the global variables) and where there is at least a *minimum number* of copies of particular thread instances (given by a function from those thread instance to $\mathbb{N}$).

Definition 24 (Control State Reachability Problem for CBP) The *control state reachability problem for CBPs* is defined as follows.
Instance: A CBP B with the initial state $init_B$ as defined page 44, a formula φ over $\mathcal{G}$ of the propositional logic, a set of thread instances L and a function $f_L : L \rightarrow \mathbb{N}$.
Question: Is there a global state $\langle \Gamma, \Lambda, \{\Theta\}\rangle$ reachable from $init_B$ such that $\Gamma(\varphi)$ and for all $I \in L$, there are at least $f_L(I)$ different thread instances I in Θ?

If the answer of the control state reachability problem on a CBP B with the initial global state $init_B$ for the boolean formula φ and the function f_L is *yes*, then we say that B *satisfies the control state reachability property for* φ *and* f_L. Otherwise, we say that B does not satisfies it.

To solve this problem, we make a reduction to the local state reachability problem on Global machines. We present such a reduction in the next section.

3.2 From CBP to the Global Machines

In this section, we explain how to translate a CBP into a GM and prove our translation rules to be correct.

Before describing this translation, we need to state an important lemma that shows that, for each command of a piece of code S, there is one and only one set of instructions remaining to be executed. Indeed, the rules that allow us to define the semantics of CBP modify the blocks of commands of thread instance (those rules apply rewriting rules on the blocks of commands of thread instances). It is thus not trivial that each time a thread instance reaches a command labelled by ℓ, the instance has the same block of commands defining the commands that remain to be executed.

This result allows us to establish a one-to-one correspondence between the pairs

(program remaining to be executed, valuation of the local variables)

which define the local states of the thread instances ; and the pairs

(program label, valuation of the local variables)

which allows us to encode the local states of the local machines. But to be complete, we need the assumption that each command of the program has a label, and that there are no two distinct commands of the program that share the same label. We enforce this thanks to the *Relabel* function defined in the next section.

Then, we eventually explain the translation.

Reachable Programs

Let $T = \langle \mathcal{V}, S \rangle$ be a thread type in a CBP B. Let us first define $\mathit{Relabel}(S)$ that transforms S by adding a new label (that is, a label that was not previously in $\mathit{Labels}(B)$) to each of its command that was not labelled. Furthermore, $\mathit{Relabel}(S)$ also adds the special instruction $[\mathbf{end}_T]\varepsilon$; at the end of S (assuming that the label $\mathbf{end}_T$ didn't exist before). In the following, we always assume that this *Relabel* function has been applied on all the thread types of CBP. For instance, we always consider in the following that we have $\mathbf{end}_T \in \mathit{Labels}(T)$ for each thread type T.

We also need to define $\widetilde{\gamma}$, an extension of γ (see Definition 21), in order to handle the special labels $[\mathbf{wait}_\ell]$ and $[\mathbf{lock}_\ell]$, that we have introduced along the axioms 3.9 and 3.7 of Section 3.1.2.

Definition 25 ($\widetilde{\gamma}$: The Extension of γ.) Let $T = \langle \mathcal{V}, S \rangle$ be a thread type and ℓ be a label. $\widetilde{\gamma}(S, \ell)$ is defined as follows.

$$\widetilde{\gamma}(S,\ell) = \begin{cases} \gamma(S,\ell) & \forall \ell \in \mathit{Labels}(B) \\ [\mathbf{wait}_{\ell'}]\mathit{wait}(M,L); \cdot S' & \text{if } \ell = \mathbf{wait}_{\ell'} \\ & \text{and } \gamma(S,\ell') = [\ell']\mathtt{sleep}(M,L)\ ; \cdot\, S' \\ [\mathbf{lock}_{\ell'}]\mathtt{lock}(L)\ ; \cdot\, S' & \text{if } \ell = \mathbf{lock}_{\ell'} \\ & \text{and } \gamma(S,\ell') = [\ell']\mathtt{sleep}(M,L)\ ; \cdot\, S' \\ \varepsilon & \text{otherwise} \end{cases}$$

$\widetilde{\gamma}$ is useful to us because, given a label ℓ and a program S, it should return exactly the part of S remaining to be executed when the label ℓ is reached. This implies that there is *one and only one* such program corresponding to *each* program label. It should be clear that this results is important to define a procedure to translate a CBP into a GM. We first have to prove that $\widetilde{\gamma}(S,\ell)$ is the only program reachable in S at label ℓ.

Let $AllLabels(T)$ for a thread type $T = \langle \mathcal{V}, S\rangle$ be the set $Labels(T) \cup \{\mathbf{wait}_\ell, \mathbf{lock}_\ell \mid \ell \in Labels(T)$ and the command labeled by ℓ in T is of the form $\texttt{sleep}(M,L)$ for some $M, L\}$.

Lemma 6 Let $B = \langle \mathcal{G}, \mathcal{C}, \{T_1, \ldots, T_i = \langle \mathcal{V}_{T_i}, S_{T_i}\rangle, \ldots, T_n\}\rangle$ be a well-formed Concurrent Boolean Program. If $\ell \in AllLabels(T_i)$ and $Init_B$ is the initial state of B, then the following holds.

$$Init_B \Rightarrow \langle \Gamma, \Lambda, \{\ldots \langle S_j, \rho_j\rangle \ldots\}\rangle \text{ with } S_j = [\ell]I \cdot S' \text{ only if } \widetilde{\gamma}(S_{T_i}, \ell) = S_j$$

Proof. We exhibit an inductive proof on the length k of the executions of the CBP.

Base case: $k = 0$. Let $\texttt{main} = \langle \mathcal{V}, S\rangle$ where $S = [\ell]I \cdot S'$. Since ℓ is not of the form $\mathbf{wait}_{\ell'}$ or $\mathbf{lock}_{\ell'}$, we obtain by definition of γ that $init_B = \langle \Gamma, \Lambda, \{\langle S, \rho\rangle\}\rangle$ with $\widetilde{\gamma}(S,\ell) = S$.

Induction step: $k = n+1$. Suppose that all the executions of length $k' \leq n$ end into a global state of the form $\langle \Gamma, \Lambda, \{E\}\rangle$ where each local state $\langle [\ell]I \cdot S', \rho\rangle \in E$ of thread type $T = \langle \mathcal{V}, S\rangle$ is such that $\widetilde{\gamma}(S,\ell) = [\ell]I \cdot S'$. Let us prove that it is also the case for the executions of length $n+1$.

Let us consider an execution of the CBP of length n that ends into the global state $G = \langle \Gamma, \Lambda, \{E\}\rangle$. Suppose that we have $G \rightarrow G'$ (G' is reachable in one step from G) with $G' = \langle \Gamma', \Lambda', \{E'\}\rangle$.

First, if Rule 3.13 (the `start` rule) is applied then a new thread instance appears in G'. Suppose that this instance is of type $T = \langle \mathcal{V}, S\rangle$ with $S = [\ell]I \cdot S'$. Hence the new thread instance is of the form $\langle S, \rho\rangle$. As ℓ is not of the form $\mathbf{wait}_{\ell'}$ or $\mathbf{lock}_{\ell'}$, we obtain by definition of γ that $\widetilde{\gamma}(S,\ell) = S$.

All the other local states in G' are constructed from a local state in G. Let us consider the local state $L \in E$ of the thread type $T = \langle \mathcal{V}, S\rangle$ with $L = \langle S_L, \rho_L\rangle$. First, if the rule applied to obtain G' from G does not change L then we have $L = L'$. Suppose that $S_L = [\ell]I \cdot S'$. We have $L' = \langle S_L, \rho_L\rangle$, and $\widetilde{\gamma}(S,\ell) = S_L$ by induction hypothesis.

If the rule applied to construct G' from G changes L into L', then the following holds. Let ℓ' be the label of the first command in L'. Let us first consider the case where ℓ' is not of the form $\mathbf{wait}_{\ell''}$ or $\mathbf{lock}_{\ell''}$ (hence we do not consider the `sleep` rule for the moment). We consider the following cases.

1. Rule 3.6 (the `goto` rule) is applied to obtain G' from G. In this case, L is replaced by a local state $L' = \langle S_{L'}, \rho_L\rangle$ such that $S_{L'}$ is of the form $[\ell']I \cdot S'$ and $S_{L'} = \gamma(S,\ell')$. Since ℓ' is not of the form $\mathbf{wait}_{\ell''}$ or $\mathbf{lock}_{\ell''}$, we have $\widetilde{\gamma}(S,\ell') = S_{L'}$;

2. A rule different from Rule 3.6, Rule 3.2 and Rule 3.4 when the condition is evaluated to $true$ (the `goto` rule, the `if` rule and the `while` rule when entering into the loop) is applied to obtain G' from G. In this case, S_L is of the form $[\ell]I \cdot [\ell']I' \cdot S''$ and L is replaced by L' in G' such that L' is of the form $\langle \rho', S_{L'}\rangle$ with $S_{L'} = [\ell']I' \cdot S''$.Let us prove that $\widetilde{\gamma}(S,\ell') = [\ell']I' \cdot S''$. In S, if ℓ and ℓ' are not imbricated into `if` and `while` blocks, then S is of the form $S_1 \cdot [\ell]I \cdot [\ell']I' \cdot S''$. The first rule of Definition 21 is applied when computing $\gamma(S,\ell')$ and we obtain $\gamma(S,\ell') = [\ell']I' \cdot S'' = S_{L'}$ and hence we have $\widetilde{\gamma}(S,\ell') = S_{L'}$ since ℓ' is not of the form $\mathbf{wait}_{\ell''}$ or $\mathbf{lock}_{\ell''}$. Remark that we could have $\ell' = [\mathbf{end}_T]$ and $S'' = \varepsilon$. In that case, since $S = S''' \cdot [\mathbf{end}_T]\varepsilon$, we have $\widetilde{\gamma}(S,\ell') = [\mathbf{end}_T]\varepsilon$ (by applying the rule 1 of Definition 21).

If ℓ and ℓ' are into `if` and `while` blocks, γ is computed recursively. Let us consider the last recursive call when computing $\gamma(S,\ell)$. At this point, we have computed $\gamma(S,\ell) = \gamma(B,\ell)\cdot S'''$ where B is of the form $B_1 \cdot [\ell]I \cdot [\ell']I' \cdot B_2$ and $S'' = B_2 \cdot S'''$. This is exactly what is computed just before the last recursive call when computing $\gamma(S,\ell')$. Indeed, as ℓ and ℓ' are in the same `if` and `while` blocks, the same rules are applied to compute γ until the last recursive call. At the last step of the computation of $\gamma(S,\ell')$, the first rule of Definition 21 is applied and we obtain that $\gamma(S,\ell') = [\ell']I' \cdot B_2 \cdot S'''$. Since $S'' = B_2 \cdot S'''$ we have $\gamma(S,\ell') = [\ell']I' \cdot S'' = S_{L'}$ and hence we have $\widetilde{\gamma}(S,\ell') = S_{L'}$ since ℓ' is not of the form $\mathbf{wait}_{\ell''}$ or $\mathbf{lock}_{\ell''}$. Remark that following the rules of Section 3.1.2, the labels $\mathbf{lock}_\ell$ and $\mathbf{wait}_\ell$ always appear in a reachable state as the label of the first command to execute by a thread;

3. Rule 3.2 (the `if` rule) or Rule 3.4 (the `while` rule) when entering into the loop is applied to obtain G' from G. Hence, $S_L = [\ell]I \cdot S'$ where the command I has one of the following forms

   - $[\ell]$`if`$(E)\{[\ell']I' \cdot S_1\}$`else`$\{S_2\}$;
   - $[\ell]$`if`$(E)\{S_2\}$`else`$\{[\ell']I' \cdot S_1\}$;
   - $[\ell]$`while`$(E)\{[\ell']I' \cdot S_1\}$.

   Suppose that it is one of the two first forms. L' is of the form $\langle \rho', [\ell']I' \cdot S_1 \cdot S'\rangle$.Let us prove that $\widetilde{\gamma}(S,\ell') = [\ell']I' \cdot S_1 \cdot S'$. As the `if` and `while` blocks containing ℓ also contain ℓ', if we have $\gamma(S,\ell) = \gamma(B,\ell) \cdot B'$ where B is of the form $B_1 \cdot [\ell]I \cdot B_2$ and $S' = B_2 \cdot B'$, then we have $\gamma(S,\ell') = \gamma(B,\ell') \cdot B' = [\ell']I' \cdot S_1 \cdot B_2 \cdot B'$. As $S' = B_2 \cdot B'$, we obtain that $\gamma(S,\ell') = [\ell']I' \cdot S_1 \cdot S'$ and hence we have $\widetilde{\gamma}(S,\ell') = [\ell']I' \cdot S_1 \cdot S'$ since ℓ' is not of the form $\mathbf{wait}_{\ell''}$ or $\mathbf{lock}_{\ell''}$.

   If I is of the third form (a `while` command where E is evaluated to *true*), then the following holds. L' is of the form $\langle \rho', [\ell']I' \cdot S_1 \cdot [\ell]I \cdot S'\rangle$.Let us prove that $\widetilde{\gamma}(S,\ell') = [\ell']I' \cdot S_1 \cdot [\ell]I \cdot S'$. As the `if` and `while` blocks containing ℓ also contain ℓ', if we have $\gamma(S,\ell) = \gamma(B,\ell) \cdot B'$ where B is of the for $B_1 \cdot [\ell]I \cdot B_2$ and $S' = B_2 \cdot B'$, then we have $\gamma(S,\ell') = \gamma(B,\ell') \cdot B' = [\ell']I' \cdot S_1 \cdot [\ell]I \cdot B_2 \cdot B'$. As $S' = B_2 \cdot B'$, we obtain that $\gamma(S,\ell') = [\ell']I' \cdot S_1 \cdot [\ell]I \cdot S'$ and hence $\widetilde{\gamma}(S,\ell') = [\ell']I' \cdot S_1 \cdot [\ell]I \cdot S'$ since ℓ' is not of the form $\mathbf{wait}_{\ell''}$ or $\mathbf{lock}_{\ell''}$.

Finally, suppose that ℓ' is of the form $\mathbf{wait}_{\ell''}$ or $\mathbf{lock}_{\ell''}$ with $\ell'' \in \mathit{Thread}(T)$. Following Rule 3.9, Rule 3.10 and Rule 3.12, to reach a command with such a label, a thread needs to first reach the command labeled by ℓ''. By induction hypothesis, we have that $\widetilde{\gamma}(S,\ell'') = [\ell'']$`sleep`$(M,L);\cdot S'$ (and $\gamma(S,\ell'') = [\ell'']$`sleep`$(M,L);\cdot S'$ since ℓ'' is not of the form $\mathbf{wait}_{\ell'''}$ or $\mathbf{lock}_{\ell'''}$) such that $[\ell'']$`sleep`$(M,L);\cdot S'$ is always the block of commands that remains to be executed when a thread instance reaches the command labelled by ℓ''. Let us consider the case where ℓ' is of the form $\mathbf{wait}_{\ell''}$. Following Rule 3.9, L' is of the form $\langle \rho, [\mathbf{wait}_{\ell''}]\mathit{wait}(M,L);\cdot S'\rangle$. Moreover, following Definition 25, $\widetilde{\gamma}(S,\ell') = [\mathbf{wait}_{\ell''}]\mathit{wait}(M,L);\cdot S'$. If ℓ' is of the form $\mathbf{lock}_{\ell''}$, then following Rule 3.10 and Rule 3.12, L' is of the form $\langle \rho, [\mathbf{lock}_{\ell''}]$`lock`$(L);\cdot S'\rangle$. Moreover, following Definition 25, $\widetilde{\gamma}(S,\ell') = [\mathbf{lock}_{\ell''}]$`lock`$(L);\cdot S'$.

In all the cases, we conclude that if a global state is reachable by an execution of length $n+1$, then all the local states L of type $T = \langle \mathcal{V}, S\rangle$ of this global state are of the form $\langle \rho, [\ell]I \cdot S'\rangle$ such that $\widetilde{\gamma}(S,\ell) = [\ell]I \cdot S'$. □

In the following, we will always consider thread instantiations $I = \langle S_I, \rho \rangle$ of thread type $T = \langle \mathcal{V}, S \rangle$ such that S_I can be constructed from S by applying $\widetilde{\gamma}$ on S, i.e. $S_I = \widetilde{\gamma}(S_I, \ell)$ for some label ℓ.

Translation

We are now assured that exactly one tuple (program label, valuation of the local variable) corresponds to each reachable local state of a thread and this will help us to define precisely the translation between the CBP and GM.

This translation is not really difficult to imagine, though there are some tricky issues. One of them is a consequence of the *dynamic creation* of threads. Such a behavior is possible in a CBP, but not in a GM, because a GM is a collection of a *fixed* number of local machines. Therefore, we have devised a special initial state for each local machine $\mathcal{L}_i$, which we denote by $\langle \theta_{\mathcal{L}_i}, \mathbf{begin}_{\mathcal{L}_i} \rangle$. This state does not correspond to any local state of a thread, and thus, any local machine in this state represents a 'not-yet-created' thread. The `start` primitive will be used to make one of these local machines *move* to the state corresponding to the initial state of the thread.

This move will represent the call to the constructor of the corresponding thread, which means that we simultaneously need to set the initial values of the local variables. We can achieve this thanks to a *rendez-vous*, whose message encodes the type of thread as well as the values to assign to the local variables.

Another problem arises from the dynamic creation of threads: in the general case, we cannot bound the number of instances of a given thread that will be created. Hence, we don't know how many local machines we should put in the corresponding global machine. We will later cope with this problem. Let us first explain how to translate each thread type of a CBP into a single local machine.

Translation of a CBP into a set of local machines. Let $\langle \mathcal{G}, \mathcal{C}, \{\langle \mathcal{V}_1, S_1 \rangle, \ldots, \langle \mathcal{V}_n, S_n \rangle\} \rangle$ be a CBP. We translate it into a *set of local machines* $\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$ such that there is a bijection between this set and the set of types of threads. The local machines will be then used to define global machines that can be viewed as having the "*same*" behaviours as the CBP.

In the sequel, we will assume that the local machine $\mathcal{L}_i$ always corresponds to the type of thread T_i.

$\mathcal{L}_i = \langle Q_i, \Sigma_i, \delta_i \rangle$ is constructed from the thread type $T_i = \langle \mathcal{V}_i, S_i \rangle$ with $|\mathcal{V}_i| = k$ as follows[4]:

1. Q_i is a set of all the elements $\langle \theta, \ell \rangle$ such that
   - θ is a valuation of the variables in $\mathcal{V}_i$ (remember that we consider that the variables in $\mathcal{V}_i$ are ordered);
   - $\ell \in \; \mathit{Labels}(T_i)$
     $\cup \{\mathbf{lock}_{\ell'}, \mathbf{wait}_{\ell'} | \exists \ell' \in \mathit{Labels}(T_i) : \gamma(S_i, \ell') \equiv [\ell']\mathtt{sleep} \cdot S_i'\}$
     $\cup \{\mathbf{begin}_{\mathcal{L}_i}\}$

   Notice that we assume here that *Labels* does not contain labels $\mathbf{lock}_\ell$, $\mathbf{wait}_\ell$, $\mathbf{begin}_{\mathcal{L}_i}$.

[4] Although the syntax of the local machines requests them, we have systematically omitted the assignments of the form $g' = g$ in the labels of the transitions in order to clarify the subsequent discussion.

2. δ_i and Σ_i can be constructed as follows. For all $\ell \in \mathit{Labels}(T_i)$, we distinguish several cases depending on the value of $\gamma(\ell, S_i)$. If it is of the form

   (a) $[\ell]u_1, \ldots, u_l{:=}v_1, \ldots v_l; \cdot [\ell'] I' \cdot S'$, then we assume a permutation of the variable indexes: $\{1, \ldots, n\} \to \{i_1, \ldots, i_n\}$, in order to *sort the variables by type*. Indeed, our permutation is such that: $\forall j:\ 1 \leq j \leq k:\ u_{i_j} \in \mathcal{V}$ and $\forall j: k < j \leq n: u_{i_l} \in \mathcal{G}$ (k may be 0). We have for each possible valuation θ of the local variables

   $$\langle \theta, \ell \rangle \xrightarrow{\tau\ : u'_{i_{k+1}} = v_{i_{k+1}} \wedge \ldots \wedge u'_{i_n} = v_{i_n}} \langle \theta', \ell' \rangle$$

   where $\theta' = \theta[u_{i_1} = v_{i_1} \ldots u_{i_k} = v_{i_k}]$.

   (b) $[\ell]\mathtt{if}(E)\ \mathtt{then}\{[\ell_1]I_1 \cdot S_1\}\mathtt{else}\{[\ell_2]I_2 \cdot S_2\} \cdot S'$, then if $E = *$, we have $\langle \theta, \ell \rangle \xrightarrow{\tau\ :\ true} \langle \theta, \ell_1 \rangle$ and $\langle \theta, \ell \rangle \xrightarrow{\tau\ :\ true} \langle \theta, \ell_2 \rangle$ for each possible valuation θ of the local variables. Otherwise, we assume that E is equivalent to the expression in disjunctive form E_1 `or` $\ldots$ `or` E_n where E_j $(1 \leq j \leq n)$ is a conjunction of terms v, $!v$ with $v \in \mathcal{V}_i \cup \mathcal{G}$. For each possible valuation θ of the local variables, we have the following. For each E_j $(1 \leq j \leq n)$, we simplify E_1 `or` $\ldots$ `or` E_n by replacing the local variables by their value in θ. If the resulting expression is $true$, then we have $\langle \theta, \ell \rangle \xrightarrow{\tau\ :\ true} \langle \theta, \ell_1 \rangle$, and if it is equivalent to $false$, we have $\langle \theta, \ell \rangle \xrightarrow{\tau\ :\ true} \langle \theta, \ell_2 \rangle$. Otherwise, let D_1 `or` $\ldots$ `or` D_m be the resulting formula where each D_i $(1 \leq i \leq m)$ is now a conjunction of terms v, $!v$ with $v \in \mathcal{G}$.
   For each D_j we have

   $$\langle \theta, \ell \rangle \xrightarrow{\tau\ :\ D_j} \langle \theta, \ell_1 \rangle$$

   This edge corresponds to the case where D_j is evaluated to true (and so is E).
   E is evaluated to false when all the D_j's are evaluated to false together. So, in addition we have (we assume here that $!!v = v$)

   $$\langle \theta, \ell \rangle \xrightarrow{\tau\ :\ \wedge_{j=1\ldots m} c_j} \langle \theta, \ell_2 \rangle$$

   such that $!c_j$ $(1 \leq j \leq m)$ appears in D_j.

   (c) $[\ell]\mathtt{choice}\{E_1{:}u_{1,1}, \ldots, u_{1,m_1}{:=}v_{1,1}, \ldots, v_{1,m_1}; \ldots E_n{:}u_{l,1}, \ldots, u_{l,m_l}{:=}v_{l,1}, \ldots, v_{l,m_l};\} \cdot [\ell'] I' \cdot S'$. To simplify the presentation, we assume that E_j $(1 \leq j \leq n)$ does not contain `or` connectors. Notice that this constraint is not limitative. Indeed, if the `choice` contains a guarded assignment E_{j1} `or` $E_{j2} : A$; (where A is an assignment), then this guarded assignment can be replaced by the two guarded assignments $E_{j1} : A$; and $E_{j2} : A$; without changing the meaning of the `choice`.

   For every guarded assignment $E_j{:}u_{j,1}, \ldots, u_{j,m_j}{:=}v_{j,1}, \ldots, v_{j,m_j}$; $(1 \leq j \leq l)$ and each possible valuation θ of the local variables, we apply the following steps: First of all we replace the local variables in E_j by their value in θ. Let D_j be the resulting formula over global variables. If $D_j = false$ then we are done for this guarded assignment (this assignment cannot be executed when the local variables have the values defined by θ). Otherwise, we consider the assignment without its guard and compute

   $$\langle \theta, \ell \rangle \xrightarrow{\tau : a} \langle \theta', \ell' \rangle$$

(where a is the action) as in the simple assignment case (see rule 2a). Now, to take the guard into account, we add D_j on the edge. Finally, we have

$$\langle \theta, \ell \rangle \xrightarrow{\tau : D_j \wedge a} \langle \theta', \ell' \rangle$$

(d) $[\ell]\mathtt{while}(E)\{[\ell_1]I_1 \cdot S_1\} \cdot [\ell_2]I_2 \cdot S_2$, then if $E = *$, we have $\langle \theta, \ell \rangle \xrightarrow{\tau \,:\, true} \langle \theta, \ell_1 \rangle$ and $\langle \theta, \ell \rangle \xrightarrow{\tau \,:\, true} \langle \theta, \ell_2 \rangle$ for each possible valuation θ of the local variables.
Otherwise, we assume that E is equivalent to the expression in disjunctive form E_1 `or` ... `or` E_n where E_j $(1 \leq j \leq n)$ is a conjunction of terms v, $!v$ with $v \in \mathcal{V}_i \cup \mathcal{G}$. For each possible valuation θ of the local variables, we have the following. For each E_j $(1 \leq j \leq n)$, we simplify E_1 `or` ... `or` E_n by replacing the local variables by their value in θ. If this simplified formula is equivalent to $true$, then we have $\langle \theta, \ell \rangle \xrightarrow{\tau \,:\, true} \langle \theta, \ell_1 \rangle$, and if it is equivalent to $false$, we have $\langle \theta, \ell \rangle \xrightarrow{\tau \,:\, true} \langle \theta, \ell_2 \rangle$.
Otherwise, let D_1 `or` ... `or` D_m be the simplified formula where each D_i $(1 \leq i \leq m)$ is now a conjunction of terms v, $!v$ with $v \in \mathcal{G}$. For each D_j $(1 \leq j \leq m)$ we have

$$\langle \theta, \ell \rangle \xrightarrow{\tau \,:\, D_j} \langle \theta, \ell_1 \rangle$$

This edge corresponds to the case where D_j is evaluated to true (and so is E).
E is evaluated to false when all the D_j's are evaluated to false together. So, in addition we have (we assume here that $!!v = v$)

$$\langle \theta, \ell \rangle \xrightarrow{\tau \,:\, \wedge_{j=1\ldots m} c_j} \langle \theta, \ell_2 \rangle$$

such that $!c_j$ $(1 \leq j \leq m)$ appears in D_j.

(e) $[\ell]\mathtt{skip}\ ; \cdot [\ell']I \cdot S''$, then for each possible valuation θ of the local variables we have

$$\langle \theta, \ell \rangle \xrightarrow{\tau \,:\, true} \langle \theta, \ell' \rangle$$

(f) $[\ell]\mathtt{goto}(\ell')\ ; \cdot S$, then for each possible valuation θ of the local variables we have

$$\langle \theta, \ell \rangle \xrightarrow{\tau \,:\, true} \langle \theta, \ell' \rangle$$

(g) $[\ell]\mathtt{lock}(L)\ ; \cdot [\ell']I' \cdot S''$, then for each possible valuation θ of the local variables we have

$$\langle \theta, \ell \rangle \xrightarrow{\tau \,:\, \neg L \wedge L' = true} \langle \theta, \ell' \rangle$$

(h) $[\ell]\mathtt{unlock}(L)\ ; \cdot [\ell']I' \cdot S''$, then for each possible valuation θ of the local variables we have

$$\langle \theta, \ell \rangle \xrightarrow{\tau \,:\, L' = false} \langle \theta, \ell' \rangle$$

because, from the well-formedness rules, at that point the lock L is already owned.

(i) $[\ell]\mathtt{sleep}(M, L)\ ; \cdot [\ell']I' \cdot S''$, then for each possible valuation θ of the local variables we have

$$\begin{cases} \langle \theta, \ell \rangle \xrightarrow{\tau : L' = false} \langle \theta, \mathbf{wait}_\ell \rangle \\ \langle \theta, \mathbf{wait}_\ell \rangle \xrightarrow{M??} \langle \theta, \mathbf{lock}_\ell \rangle \\ \langle \theta, \mathbf{wait}_\ell \rangle \xrightarrow{M\downarrow} \langle \theta, \mathbf{lock}_\ell \rangle \\ \langle \theta, \mathbf{lock}_\ell \rangle \xrightarrow{\tau \,:\, \neg L \wedge L' = true} \langle \theta, \ell' \rangle \end{cases}$$

Notice that in the case of the first transition, the well-formedness rules ensure that the thread has the lock L when it executes the command `sleep`(M, L) for some message M and hence it can release it. Moreover, we need the second and the third transition as a thread can be awoken either by a `wakeup` or by a `wakeupall`.

(j) $[\ell]$`wakeup`(M) ; $\cdot\, [\ell']I' \cdot S''$, then for each possible valuation θ of the local variables we have

$$\langle \theta, \ell \rangle \xrightarrow{M\uparrow \,:\, true} \langle \theta, \ell' \rangle$$

(k) $[\ell]$`wakeupall`(M) ;$\cdot[\ell']I' \cdot S''$, then for each possible valuation θ of the local variables we have

$$\langle \theta, \ell \rangle \xrightarrow{M!! \,:\, true} \langle \theta, \ell' \rangle$$

(l) $[\ell]$`start`$(T, \pi_1, \ldots, \pi_n)$; $\cdot\, [\ell']I' \cdot S''$, then for each possible valuation θ of the local variables we have

$$\langle \theta, \ell \rangle \xrightarrow{T\pi_1 \ldots \pi_n! \,:\, true} \langle \theta, \ell' \rangle$$

(m) $[\ell]$`rendezvous`$(M, u_1, \ldots, u_l) \cdot [\ell']I \cdot S'$, then for each possible valuation θ of the local variables we have

$$\langle \theta, \ell \rangle \xrightarrow{M\theta(u_1) \ldots \theta(u_l)! \,:\, true} \langle \theta, \ell' \rangle$$

(n) $[\ell]$`accept`$(M, v_1, \ldots, v_l)$; $\cdot\, [\ell']I' \cdot S''$, then for each possible valuation θ of the local variables we have

$$\forall \mathbf{a} = \langle a_1, \ldots, a_l \rangle \in \{true, false\}^l : \; \langle \theta, \ell \rangle \xrightarrow{M\mathbf{a}?} \langle \theta[v_1 = a_1, \ldots, v_l = a_l], \ell' \rangle$$

Finally, assuming that $S_i = [\ell]I \cdot S'$, we have

$$\forall \mathbf{a} = \langle a_1, \ldots, a_k \rangle \in \{true, false\}^k : \langle \theta_{\mathcal{L}_i}, \mathbf{begin}_{\mathcal{L}_i} \rangle \xrightarrow{T_i\mathbf{a}?} \langle \theta, \ell \rangle$$

where for all $1 \leq j \leq k$, we have $\theta(v_j) = a_j$.

3. Σ_i is constructed together with δ_i. More precisely, we construct the set Δ_i of labels on which Σ_i is built as follows:

   (a) Each time we encounter a message M as a parameter of a `wakeup`() ;, `wakeupall`() ; or `unlock`() ; during the construction of δ_i, we add M to Δ_i ;

   (b) For each transition $\langle \theta, \ell \rangle \xrightarrow{Ma_1 \ldots a_l?} \langle \theta, \ell' \rangle$ added to δ_i, we add to Δ_i the message $Ma_1 \ldots a_l$;

   (c) Finally, for each thread type name T where the threads of type T have k variables, we add the set $\{Ta_1 \ldots a_k | \langle a_1, \ldots, a_k \rangle \in \{true, false\}^k\}$ of messages to Δ_i.

The local machines constructed following this translation satisfy the constraints of Definition 3. Indeed, the source local state of a broadcast or a signal always corresponds to a $wait_\ell$ command and the target local state of a broadcast or a signal always corresponds to a `lock` command (see Rule (2i)) and hence a local state cannot be the source of broadcast or a signal and the target local state of a broadcast or a signal. This ensures that the first (avoiding cyclic behaviours) are satisfied by the local machines that are constructed by the translation. Moreover, all the edges leaving a source local state of a broadcast or a signal go to the same target local state (see Rule (2i)). So, the second constraint is also guaranteed to be satisfied.

Using our translation, we can now easily obtain a set of local machines from a given CBP. But this is not enough to construct a full GM. Indeed, we need to specify how many instances of each local machine exist in the GM. As each local machine corresponds to one thread instantiation, and as we cannot bound the number of threads that will be instantiated, we define the result of the translation of the CBP as a *family of Global Machines*. It will be the set of all the global machines constituted of any number of the previously obtained local machines.

Definition 26 (Resulting Family of Global Machines.) Given a CBP $B = \langle \mathcal{G}, \mathcal{C}, \mathcal{T} \rangle$ and its translation into a set of local machines $\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$, $\mathcal{F}(B)$ is a set of global machines, called the *Family of Global Machine of* B, defined as follows.

$$\mathcal{F}(B) = \{\langle \mathcal{G} \cup \mathcal{C}, \{\langle \mathcal{L}_1, k_1 \rangle, \ldots, \langle \mathcal{L}_n, k_n \rangle\}\rangle \text{ such that } \forall 1 \leq i \leq n \ : \ k_i \geq 1\}$$

Example 3 *The translation of the CBP of Figure 3.7 into a set of local machines is represented in Figures 3.3 and Figure 3.4. In those examples, we removed the internal action labels and only kept the rendez-vous labels and broadcast labels (there are no signal labels in those examples). We also replaced the negation "!" by "not" to distinguish a rendez-vous from a test. Finally, we omitted the assignment of the form* $x' = x$.

In the following we only consider runs of GMs with an initial global state corresponding to the initial state of the CBP. More precisely, let $G = \langle \mathbf{B}, \{\langle \mathcal{L}_1, k_1 \rangle, \ldots, \langle \mathcal{L}_n, k_n \rangle\}\rangle$ be a GM constructed from the CBP $B = \langle \mathcal{G}, \mathcal{C}, \{T_1 = \langle \mathcal{V}_1, S_1 \rangle, T_2, \ldots, T_n\}\rangle$ where T_1 corresponds to the `main` thread and $S_1 = [\ell] I \cdot S$. We only consider runs with an initial global state of the form $\langle \rho, \langle s_{11}, \ldots, s_{1k_1}, \ldots, s_{n1}, \ldots, s_{nk_n} \rangle \rangle$ where (i) for all $v \in \mathcal{G}$ we have $\rho(v) = true$, (ii) $s_{11} = \langle \rho', \ell \rangle$ where for all $v \in \mathcal{V}_1$ we have $\rho'(v) = true$, and (iii) for all $s_{kl} \neq s_{11}$ we have $s_{kl} = \langle \theta_{\mathcal{L}_k}, \mathbf{begin}_{\mathcal{L}_k} \rangle$.

3.2.1 Relation between the CBPs and the GMs

It is now time to show that our translation from the CBP to the GM preserves the semantics, more precisely that it is trace-conservative. Therefore, we prove the *trace inclusion* between the CBPs and the GMs, in both directions. Roughly speaking: given a CBP and its translation into a family of GM : (i) for each finite execution of the CBP, there exists a run of a GM of the family with an equivalent trace (in the sense of the mapping functions), and (ii), for every run of a GM of the family, there exists an execution of the CBP with an equivalent trace. This means that we can safely verify on the GM the properties on finite runs we are interested in; and immediately deduce the corresponding property on the CBP.

Before we can state the proof of this statement, we need to define some *mapping functions* to establish a correspondence between the global and local states of both models. Our set of mapping functions is as follows: the χ function maps the thread instantiations on tuples describing a local state of a local machine and the Φ function maps a CBP's global state on a tuple describing a global state of a global machine.

Recall that we consider CBPs where all the commands are labelled. The χ and Φ functions are defined as follows.

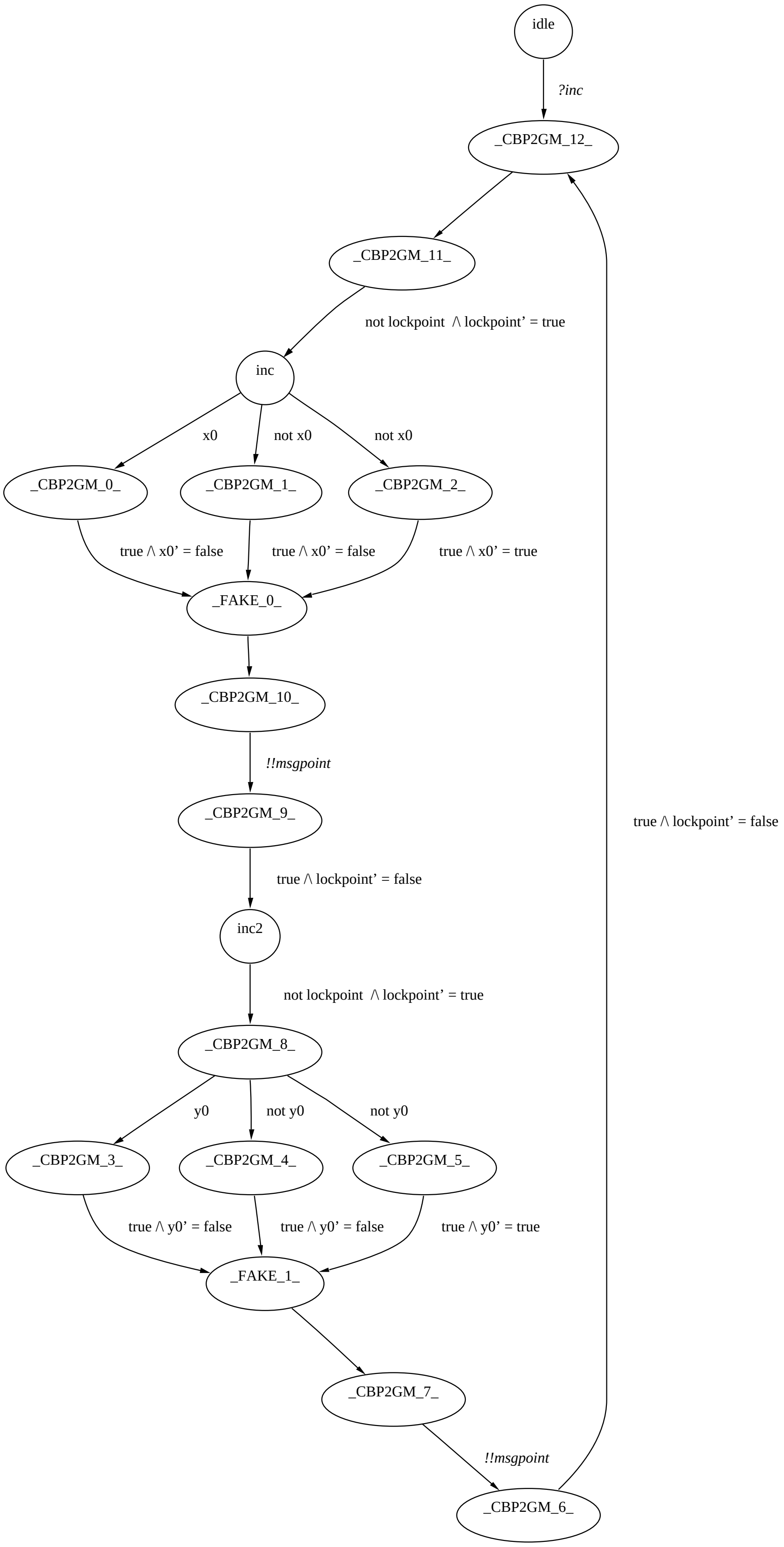

Figure 3.3: The Local Machine for the thread type `Inc`.

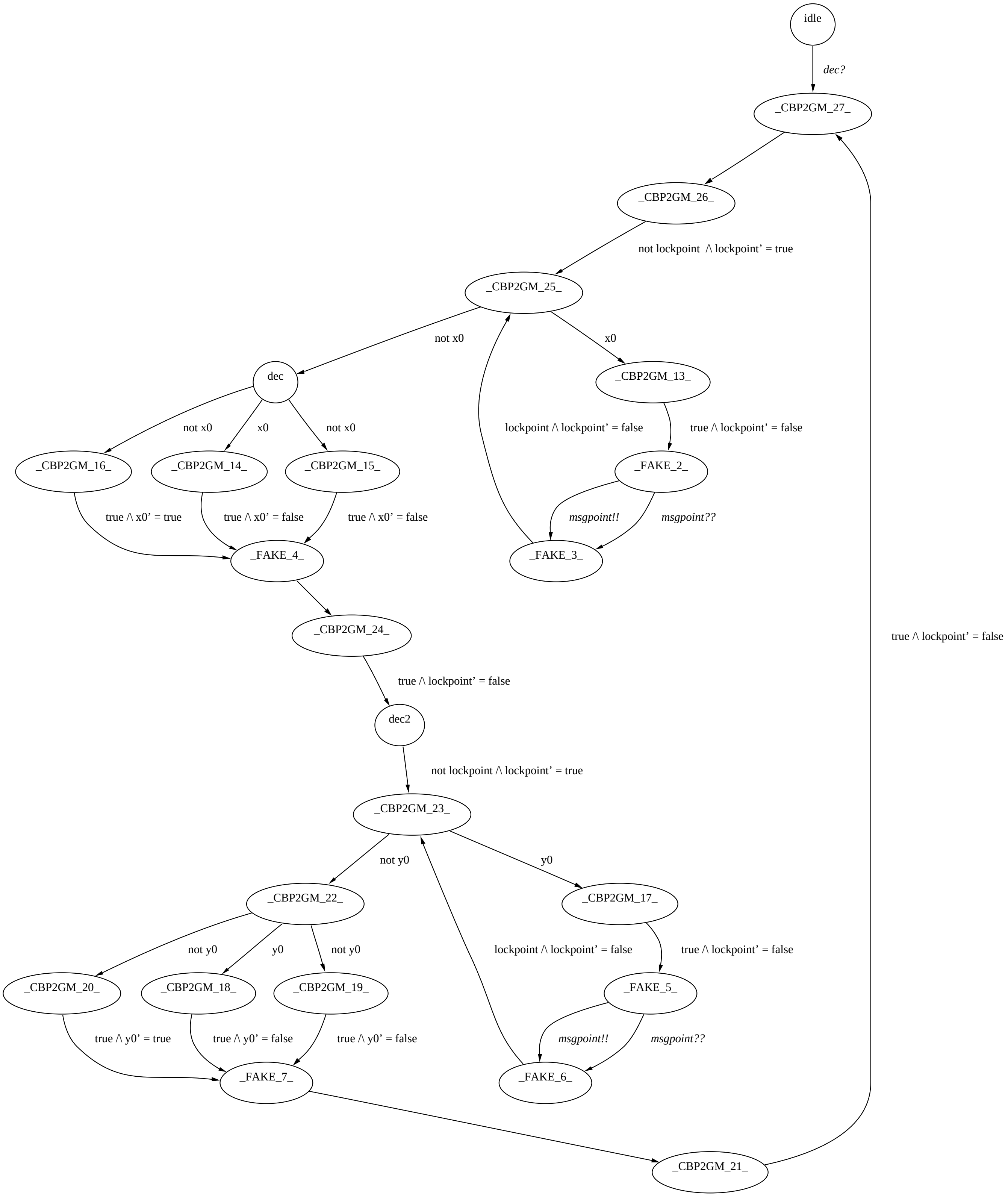

Figure 3.4: The Local Machine for the thread type `Dec`.

Definition 27 (The χ Function) Given a thread instantiation $I = \langle S, \rho\rangle$ with $S = [\ell]I \cdot S'$, $\chi(I)$ is defined as the tuple $\langle \rho, \ell\rangle$.

Definition 28 (The Φ Function) Given a global state $S = \langle \Gamma, \Lambda, \{I_1, \ldots, I_n\}\rangle$ of a CBPB, $\Phi(S)$ is defined as the tuple $\langle \Gamma \cup f(\Lambda), \{\chi(I_1), \ldots, \chi(I_2)\}\rangle$.

The f function is defined as follows. For every lock l we have

$$f(\Lambda)(l) = \begin{cases} true & \text{if } \Lambda(l) = locked \\ false & \text{if } \Lambda(l) = unlocked \end{cases}$$

$\Phi(S)$ can be viewed as the global state corresponding to S in a particular GM $\mathcal{G}$ in $\mathcal{F}(B)$. $\mathcal{G}$ is such that the number of local machines $\mathcal{L}_i$ in $\mathcal{G}$ is equal to the number of thread instances of type T_i (the thread type corresponding to $\mathcal{L}_i$) in S. $\Phi(S)$ can also be viewed as a global state of GMs in $\mathcal{F}(B)$ where we have more local machines than in $\mathcal{G}$ but where the exceeding local machines are in their initial local state ($\langle \theta_{\mathcal{L}_i}, \mathbf{begin}_{\mathcal{L}_i}\rangle$ for a machine $\mathcal{L}_i$) and have been forgotten. More precisely, $\Phi(S)$ can be viewed as a global state of GMs in the corresponding family of B where the local machines that are in their initial local state (and so do not correspond to active thread instances in S) have been forgotten.

Φ is extended to (finite) sequences of global states of CBP as follows. Given a sequence $S_1 S_2 \ldots S_n$ of global states of the CBP B, we define $\Phi(S_1 S_2 \ldots S_n)$ such that $\Phi(S_1 S_2 \ldots S_n) = \Phi(S_1)\Phi(S_2)\ldots\Phi(S_n)$.

There is a last problem we have to cope with. We should indeed take into account the 'not-yet-created' threads, that appear in the GM as local machines in the special $\langle \theta_{\mathcal{L}_i}, \mathbf{begin}_{\mathcal{L}_i}\rangle$ states. The *Created* function allows us to 'forget about' the non-active local machines in a global state of a GM, and focus only on the ones that have a real corresponding thread in the global state of the CBP.

Definition 29 (The *Created* Function) Given a global state $G = \langle \rho, \{s_1, \ldots, s_n\}\rangle$ of a $GM = \langle \mathbf{B}, \langle \mathcal{L}_1, k_1\rangle, \ldots, \langle \mathcal{L}_m, k_m\rangle\rangle$, $Created(G) = \langle \rho, \{created(s_1) + \ldots + created(s_n)\}\rangle$ where

$$created(\langle \theta, \ell\rangle) = \begin{cases} \{\langle \theta, \ell\rangle\} & \text{if } \ell \not\equiv \mathbf{begin}_{\mathcal{L}_i} \text{ for some } 1 \le i \le m; \\ \emptyset & \text{otherwise.} \end{cases}$$

Created is extended to runs of GM as follows. Given a run $G_1 \ldots G_n$ of a GM $\mathcal{G}$, we define $Created(G_1 \ldots G_n)$ such that $Created(G_1 \ldots G_n) = Created(G_1) \ldots Created(G_n)$.

Our last auxiliary function is NT, a counting function, that will turn to be useful in the sequel.

Definition 30 (Number of Threads into a Global State) Given a global state G of a CBP B, and a type of thread T_i, $NT(G, T_i)$ is defined as the number of thread instantiations of type T_i in G.

We can now eventually state the lemmas we claimed. Note the restriction on the number of thread instantiations. Indeed, one cannot map an execution of a CBP with n threads of a given type T, onto a run of a GM that has less than n local machines corresponding to T.

Lemma 7 (Trace Inclusion: From CBP to GM) Given a $CBP\ B = \langle \mathcal{G}, \mathcal{C}, \{T_1, \ldots, T_n\}\rangle$, a GM $\mathcal{G}_B \in \mathcal{F}(B)$ with $\mathcal{G}_B = \langle \mathbf{B}, (\mathcal{L}_1, k_1), \ldots, (\mathcal{L}_n, k_n)\rangle$ and a finite execution e of B, there exists a finite run R of $\mathcal{G}_B$ such that:

$$\Phi(e) = Created(R)$$

provided that the number of threads of each type in e never exceeds the number of instances of the corresponding local machine. More precisely, for every global state G in e, we have: $\forall 1 \leq i \leq n \ : \ NT(G, T_i) \leq k_i$.

Proof. The proof is given in Appendix. □

In the other direction, the inclusion of the runs of a GM into the executions of the corresponding CBP can be proved similarly.

Lemma 8 (Trace Inclusion: From GM to CBP) Given a CBP $B = \langle \mathcal{G}, \mathcal{C}, \{T_1, \ldots, T_n\} \rangle$, a GM $\mathcal{G}_B \in \mathcal{F}(B)$ with $\mathcal{G}_B = \langle \mathbf{B}, (\mathcal{L}_1, k_1), \ldots, (\mathcal{L}_n, k_n) \rangle$, and a finite run R of $\mathcal{G}_B$, there exists a finite execution e of B such that:

$$\Phi(e) = Created(R)$$

Proof. To prove this lemma, one can apply the same reasoning that was previously used for Lemma 7. □

Thanks to the previous lemmas, we can now state the following equivalence theorem.

Theorem 6 (Trace Equivalence) Let $B = \langle \mathcal{G}, \mathcal{C}, \{T_1, \ldots, T_n\} \rangle$ be a CBP and $\mathcal{F}(B)$ be its corresponding family of GM.

Then, e is an execution of B iff there exists a GM $\mathcal{G}_B \in \mathcal{F}(B)$, and a run R of $\mathcal{G}_B$ such that:

$$\Phi(e) = Created(R)$$

Proof. $\Rightarrow$ Let e be an execution where we have k_i thread instances T_i $(1 \leq i \leq n)$. Let $\mathcal{G}_B \in \mathcal{F}(B)$ be such that $\mathcal{G}_B$ is composed by k_i local machines of type $\mathcal{L}_i$ $(1 \leq i \leq n)$. Following Lemma 7, we have that there exists a run R of $\mathcal{G}_B$ such that $\Phi(e) = Created(R)$.

$\Leftarrow$ Let R be a run of $\mathcal{G}_B \in \mathcal{F}(B)$. Following Lemma 8, we have that there exists an execution e of B such that $\Phi(e) = Created(R)$. □

Theorem 6 allows us to reduce the control state reachability problem on CBPs to the local state reachability problem on GMs. More precisely, for a CBP B, a formula φ on the global variables of B, a set L of thread instantiations and a function $f_L : L \rightarrow \mathbb{N}$, the control state reachability problem on B for φ and f_L can be decided by finding a global machine $\mathcal{G} \in \mathcal{F}(B)$ satisfying the local state reachability property for φ and the function f. f is defined as follows. Let LS be the set of all the local states of all the local machines composing the global machines of $\mathcal{F}(B)$. For each $s \in LS$, if there exists $I \in L$ such that $\chi(I) = s$, then $f(s) = f_L(I)$, otherwise $f(s) = 0$ (notice that as the initial states of local machines do not correspond to thread instances, we always have $f(\langle \theta_{\mathcal{L}_i}, \mathbf{begin}_{\mathcal{L}_i} \rangle) = 0$ for any local machine $\mathcal{L}_i$).

Formally, we have the following.

Corollary 3 The CBP B satisfies the control state reachability property for the function f_L and the formula φ if and only if there exists $\mathcal{G} \in \mathcal{F}(B)$ that satisfies the local state reachability property for the formula φ and the function f.

This result is particularly interesting. Indeed, all the global machines $\mathcal{G}_B \in \mathcal{F}(B)$ are built from the same set of thread types and the same set of CBP global variables, and so contain the same types of local machines and the same set of global variables. As a consequence, a family of global machines can be viewed as a global machine where the number of instances of each types of local machine is a parameter. For instance, the family built from the global variables $\mathbf{B}$ and the set of local machines $\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$ is represented by the parametric global machine $\langle \mathbf{B}, \langle \mathcal{L}_1, \omega_1 \rangle, \ldots, \langle \mathcal{L}_n, \omega_n \rangle \rangle$ where ω_i $(1 \leq i \leq n)$ is a parameter meaning that we can have any number of copies of $\mathcal{L}_i$. We associate to this parametric global machine the parametric global state G_0 where all the global variables are equal to *true* and the set of local states is composed of ω_i instances of the local state $\langle \theta_{\mathcal{L}_i}, \mathbf{begin}_{\mathcal{L}_i} \rangle$ for $1 \leq i \leq n$ and one instance of the local state $\langle \rho, \ell \rangle$ such that $\langle \mathcal{V}, [\ell] I \cdot S \rangle$ is the main thread type and $\rho(x) = true$ for all $x \in \mathcal{V}$. When instantiating the parameters, we obtain the set of global machines in $\mathcal{F}(B)$ with their initial global state.

This is also important for the translation of global machines to MTN. The application of the translation of Section 2.3 on each $\mathcal{G}_B \in \mathcal{F}_B$ will always return the same MTN, i.e. the same set of places and the same set of multi-transfers are constructed when applying the translation on the global machines of $\mathcal{F}_B$. Actually, only the initial markings that are considered for each MTN will change. Those markings only differs in the number of tokens in the places corresponding to the initial local states of local machines. So, we apply the translation once and we associate a parametric initial marking to the resulting MTN. Parameters in the initial marking describes the fact that we can have any number of tokens in the places that correspond to initial local state (in fact, this marking is $\alpha_{\mathcal{M}_\mathcal{G}}(G_0)$ (see Section 2.3) where G_0 is the parametric initial global state associated to the parametric GM). This parametric MTN finitely describes the infinite set of MTN plus initial marking obtained by applying the translation on the global machines of the family of global machines associated to B.

The local state reachability problem on global machines and the coverability problem on MTN are naturally extended to the parametric case.

More precisely, the local state reachability is extended as follows. Let $\mathcal{G}$ be a parametric global machine with a parametric initial global state G_0, φ a formula over $\mathbf{B}$ of the propositional logic and $f_S : Q_1 \cup \ldots \cup Q_m \to \mathbb{N}$. The local state reachability on $\mathcal{G}$ and G_0 for φ and f_S consists to decide if there exists a global machine with an initial global state constructed from $\mathcal{G}$ and G_0 by instantiating the parameters (i.e. a global machine of $\mathcal{F}(B)$) and that satisfies the local state reachability for φ and f_S. The coverability problem on MTN is extended to the parametric case in the same way.

Moreover, the reduction of the local state reachability problem to the coverability problem is extended to the parametric case as follows. Let $\mathcal{G} = \langle \mathbf{B}, \langle \mathcal{L}_1, \omega_1 \rangle, \ldots, \langle \mathcal{L}_k, \omega_n \rangle \rangle$ be a parametric global machine and $\mathcal{M}_\mathcal{G}$ be the parametric MTN constructed from $\mathcal{G}$ by applying the translation presented in Section 2.3. We have that $\mathcal{G}$ with the initial parametric global state G_0 satisfies the local state reachability property for the formula φ of the propositional logic and the function f_S if and only if $\mathcal{M}_\mathcal{G}$ with the initial parametric marking $\alpha_{\mathcal{M}_\mathcal{G}}(G_0)$ (the initial marking corresponding to G_0) satisfies the coverability property for S where S is defined as in the non-parametric case (see Theorem 5).

In Chapter 5, we show how to deal with parameters to verify properties on a parametric

MTN.

3.3 Verification of JAVA Programs

In this section, we propose a general approach for the verification of multi threaded JAVA programs.

First, we show how to construct a CBP from JAVA programs using predicate abstraction. Thanks to the automatic construction of MTNs from CBPs presented previously in this thesis, this allows us to use the automatic verification procedures on MTNs presented in Chapter 5 to verify JAVA programs. As the process of construction of CBPs sometimes leads to coarser models on which the desired properties cannot be verified, we then propose a general approach allowing to avoid this problem in practice.

3.3.1 From JAVA Programs to CBPs

Under some assumptions, multi threaded JAVA programs can be encoded into CBPs. We present in this section a simple technique to abstract JAVA using the well-known predicate abstraction. This technique has been already defined and studied in many works (see for instance [GS97, BMMR01, BPR02, HJMS02]). So, we informally present here how to apply predicate abstraction on JAVA programs to obtain CBPs. In the following, we will assume that

1. the global objects exist when they are accessed by threads. This simplification can be easily avoided by adding some information on the state of the objects (the object exists or not) in the abstract models.

2. the global objects (objects accessed by several threads) can be detected using static analysis techniques. This means that we will not focus on the problem of detecting global objects in the following.

3. there is a finite number of global objects. In section 3.3.2, we will show how to avoid this restriction.

4. the threads have a finite control part, i.e. we avoid infinite recursions. This means that we can always *inline* the methods and see each `run` method of threads as one block without method calls inside.

5. the threads use references to another thread objects only to create and start new ones. This means that the threads do not communicate by method calls or by accessing directly to local variables of other threads. This restriction ensures that the identities of threads are not relevant to verify properties on multi-threaded JAVA programs.

These assumptions could be seen as restrictive but, as shown on particular examples, some of the restrictions needed here could be avoided by using more complex abstraction techniques. The encoding of JAVA programs into CBPs works as follows. The notifyAll primitive of JAVA is encoded with the broadcast operation of CBPs, the notify primitive is encoded with the signal operation and sleep is used for the wait primitive. The start operation of CBPs is used for instantiations of threads. The locks of objects are translated into locks of CBPs and the lock and unlock operations are used to manage the synchronized methods (blocks) of JAVA. Variables of

JAVA threads are encoded using the *predicate abstraction* technique. This technique, fully automatisable [BHPV00, BMMR01] even when predicates relate to object references [BMR02], is used to construct finite abstractions of infinite state systems. This technique was first proposed by S. Graf and H. Saidi in [GS97] and recently applied in [BMMR01, BPR02, HJMS02] to prove safety properties of software. Our approach differs from the previous works in the sense that the resulting (abstract) programs still have a (potentially) infinite state space as the number of threads may be unbounded.

Mainly, predicate abstraction is based on the fact that only the information that allows us to decide the tests in the if and while statements is relevant to define the set of executions of the programs. A boolean variable b_i is defined for each condition e_i in the program and define the truth value of that expression e_i, i.e. $b_i = true$ expresses that the corresponding constraint e_i is evaluated to true. JAVA programs are translated into concurrent boolean programs having the same control part. This means that the abstract programs are programs with the boolean variables b_i's and where the operations describe the modifications of the truth value of the expressions e_i when applying the concrete program operations.

In CBPs, global variables correspond to predicates on data's of global objects and local variables correspond to predicates referring to local data's.

```
public class Point{
private int x = 0;
private int y = 0;
public synchronized void incx(){
  x = x + 1;
  notifyAll();}
public synchronized void decx() {
  while (x == 0)
    wait();
  x = x - 1;}
}
```

```
public synchronized void incy(){
  y = t + 1;
  notifyAll();}
public synchronized void decy() {
  while (y == 0)
    wait();
  y = y - 1;}
```

Figure 3.5: A simple example of JAVA class

```
public class Inc extends Thread{
private Point p;
public Inc(Point p){
  this.p = p;}
private void incpoint(){
  p.incx();
  p.incy();}
public void run(){
  while (true) incpoint();}
}
```

```
public class Dec extends Thread{
private Point p;
public Dec(Point p){
  this.p = p;}
private void decpoint(){
  p.decx();
  p.decy();}
public void run(){
  while (true) decpoint();}
}
```

Figure 3.6: JAVA thread classes manipulating a shared Point object

As an example, let us look at the JAVA thread classes of Figure 3.6 and let us consider a program composed of some such threads. These threads manipulate a shared Point object described in Figure 3.5. After the creation of a Point object that will be shared by all the threads of the program, threads either increment or decrement coordinates of this Point object using the synchronized methods incx, decx, etc ... Coordinates never go down zero and when a thread tries to decrement coordinates equal to zero, it starts waiting (applying the wait primitive) until the coordinates go up to zero. After incrementing the coordinates, threads execute the notifyAll method to wake up all the potentially waiting threads. These awaken threads compete for the lock of the Point object and, then, either decrement a coordinate of the shared object or start waiting again. The first case occurs when no other threads put coordinates to zero again before the awaken thread takes the lock of the Point object and blocks the access to its coordinates. The second case occurs when other threads put coordinates to zero again before the awaken thread takes the lock.

To translate these JAVA threads into threads of CBP, first at all, we inline the methods of the Point class into the run method of the Inc class and the Dec class.

Then, the predicate abstraction technique is applied on the inlined program. In our example, there are two loops guarded by the constraints x == 0 and y == 0. Two boolean variables $x0$ and $y0$ are defined and describe the truth value of the two constraints respectively. More precisely, the fact that $x0$ is equal to zero means that the expression x == 0 is true.

The CBP program is constructed by replacing the tests and operations on x and y by tests and operations on $x0$ and $y0$. The conditions of the while statements in the decx and decy methods are replaced by $x0$ and $y0$, respectively. The operations $x = x + 1$; of the JAVA program are replaced by the operation

```
choice {
  x0 : x0 := false;
  !x0 : x0 := false;
  !x0 : x0 := true;
}
```

This means that if x is equal to zero before the increment, it will be different from 0 after the increment. In the other case, x is different from zero before the increment and x can be either equal to zero (if x was equal to -1) or not (if we had $x \geq 1$ or $x \leq -2$ before the increment). The increment operations on y are translated in the same way.

The operations $x = x - 1$; of the JAVA program are translated into the operation

```
choice {
  x0 : x0 := false;
  !x0 : x0 := false;
  !x0 : x0 := true;
}
```

This means that if x is equal to zero before the decrement, it will be different from zero after it. Otherwise, x is different from zero before the decrement and either x becomes equal to zero or remains different from zero. The operations $y = y - 1$; are abstracted in the same way.

Non determinism is a consequence of the loss of information on the values of variables during the translation from JAVA programs to CBPs. As the exact values of the concrete variables are lost, it is not always possible to define from an abstract state a unique successor state resulting in the application of a transformation or a test.

This means that, sometimes, the CBP constructed from the JAVA program will make choices that are not relevant during its executions with respect to the JAVA program, i.e. some executions of the CBP do not correspond to an execution of the JAVA program. However, each execution of the JAVA program corresponds to an execution of the CBP. So, if the set of executions of the CBP satisfies some properties, so does the set of executions of the JAVA program. However, due to the (over) approximation of the abstraction, the converse does not hold.

However, it is reported in [BMMR01] that the loss of information is often negligible in practice to verify software programs (and systems in a more general way).

Finally, a lock lockpoint and a message msgpoint are used, respectively, to simulate the synchronized blocks and the wait/notifyAll mechanism.

Figure 3.7 shows the CBP threads corresponding to the JAVA threads of Figure 3.6.

```
inc {vars : ;
  while(true){
    lock(lockpoint);
    choice {
      x0 : x0 := false;
      !x0 : x0 := false;
      !x0 : x0 := true;
    }
    wakeupall(msgpoint,lockpoint);
    unlock(lockpoint);
    lock(lockpoint);
    choice {
      y0 : y0 := false;
      !y0 : y0 := false;
      !y0 : y0 := false;
    }
    wakeupall(msgpoint,lockpoint);
    unlock(lockpoint);
  }
}
```

```
dec {vars : ;
  while(true){
    lock(lockpoint);
    while(x0){
      sleep(msgpoint,lockpoint);}
      choice {
        x0 : x0 := false;
        !x0 : x0 := false;
        !x0 : x0 := true;
      }
    unlock(lockpoint);
    lock(lockpoint);
    while(y0){
      sleep(msgpoint,lockpoint);}
    choice {
      y0 : y0 := false;
      !y0 : y0 := false;
      !y0 : y0 := true;
    }
    unlock(lockpoint);
  }
}
```

Figure 3.7: The CBP threads corresponding to the JAVA threads of Figure 3.6

In the next section, we present some examples of abstraction of JAVA programs. More precisely, we present on several examples taken from the literature [Lea00] a methodology to abstract JAVA programs into MTNs using the predicate abstraction presented in this section. These MTNs can then be used to verify the programs using the algorithms presented in Chapter 5. Although the abstraction phases of JAVA programs into CBPs is out of the scope of this thesis, we informally present for each example the basic ideas of our approach.

3.3.2 Examples of abstraction of JAVA programs

In this section, we present three interesting examples of JAVA programs and show how to abstract them into CBP.

Example 1: The account management program example

We consider in this section an application managing bank accounts. Clients of the bank can deposit or withdraw money on their accounts in the usual way. The system distinguishes two types of account, the *checking accounts* and the *saving accounts*. An automatic transfer service between the checking account and the saving account of clients is proposed. More precisely, an amount of money is automatically transfered from the saving account to checking account of a client when the checking account is under a lower bound and there is enough money in the saving account.

```
public class Checkingaccount{
protected Savingaccount sav;
protected long balance;
long threshold;
public Checkingaccount(long t){
  threshold=t;
  balance=0;}
public Initsaving
       (Savingaccount saving){
  this.sav=saving;}
public synchronized long balance(){
  return balance;}
public synchronized boolean
       deposit(long amount){
  if (balance+amount<0)
    return false;
  else{
    balance+=amount;
    synchronized(sav){
    if (balance<threshold &&
    sav.balance()>=sav.maxtransfer){
      sav.withdraw(sav.maxtransfer);
      deposit(sav.maxtransfer);}
    }
  }
  return true;}
public synchronized boolean
       withdraw(long amount) {
  return deposit(-amount);}
}
```

```
public class Savingaccount{
protected Checkingaccount check;
protected long balance;
long maxtransfer;
public Savingaccount(long max){
  maxtransfer=max;
  balance=0;}
public Initchecking
       (Checkingaccount checking){
  this.check=checking;}
public synchronized long balance(){
  return balance;}
public synchronized
       deposit(long amount){
  if(balance+amount<0)
    return false;
  else {
    balance+=amount;
    synchronized(check){
    if (balance>=maxtransfer &&
    check.balance()<check.threshold){
      withdraw(maxtransfer);
      check.deposit(maxtransfer);}
    }
  }
  return true;}
public synchronized boolean
       withdraw(long amount) {
  return deposit(-amount);}
```

Figure 3.8: A simple solution for the automatic transfer service

The simplest way to implement this feature is shown in Figure 3.8. Two kinds of objects are used.

The first ones are instantiations of the Checkingaccount class. When an account of this kind goes down a threshold (defined by the threshold variable) after a withdraw operation and there is enough money in its saving account, i.e. the balance of the saving account is greater or equal to maxtransfer, the amount of money maxtransfer is removed from the saving account and deposited into the checking account.

The second class is the Savingaccount class. The automatic transfer in the saving account point of view is implemented as follows. Each time some money is deposited into the account, an amount of money maxtransfer is transfered to the corresponding checking account if the balance of this account is under its threshold and there is enough money into the saving account.

We consider in the following a program composed by the thread classes shown in Figure 3.9. The instantiations of those classes access (non-deterministically) to Checkingaccount objects and Savingaccount objects through two global Vector objects Vcheck (that contains the Checkingaccount objects) and Vsav (that contains the Savingaccount objects). The main method of the program only creates Savingaccount objects and Checkingaccount objects (the exact number is chosen non-deterministically), and then creates CheckingThread threads and Savingthread threads. As in the cases of the accounts, the exact number of threads is chosen non-deterministically.

To provide an exclusive access on the Checkingaccount and Savingaccount objects, locks are used via synchronized methods.

At some point of their execution, threads can have the lock of a checking account and tries to acquire the lock of the corresponding saving account and vice versa. In that case, a critical property that the system must ensure is the *deadlock freedom* property i.e. it must ensure that there are never two threads waiting for a lock taken by the other one. Figure 3.11 shows how a thread A and a thread B fall into a deadlock situation. First, the thread A takes $lock_1$ and the thread B takes $lock_2$ (Figure 3.11(a)). Then, the thread A tries to take $lock_2$ and waits for its availability (Figure 3.11(b)). Finally, the thread B tries to take $lock_1$ and starts waiting forever (Figure 3.11(c)).

The application manages an unbounded number of accounts. Since CBPs have only a finite number of global variables and locks, the program must be aggressively abstracted. Actually, only information on a particular checking account $\mathcal{C}$ and its saving account $\mathcal{S}$ is kept, i.e. only the locks and predicates on the data's of $\mathcal{C}$ and $\mathcal{S}$ remain in the abstract program. This leads to a bigger over-approximation than when information on all the possible accounts is kept. As $\mathcal{C}$ and $\mathcal{S}$ are chosen without assumption, then the following holds.

In the abstract program, if there is never a thread A having the lock of $\mathcal{C}$ and waiting for the lock of $\mathcal{S}$ while there is a thread B having the lock of $\mathcal{S}$ and waiting for the lock of $\mathcal{C}$, then there are never two threads A and B in the JAVA program such that A has the lock of a checking account and waits for the lock of the corresponding saving account while B has the lock of the saving account and waits for the lock of the corresponding checking account.

Figure 3.10 shows the CBP threads that withdraw money into the checking accounts and deposit money into the saving accounts. These abstract threads are constructed from the JAVA threads of Figure 3.8 following the principles of Section 3.3.1 using no predicate. The if branch in the thread Checkingthread (Savingthread) corresponds to an access to $\mathcal{C}$ ($\mathcal{S}$). The else branch corresponds to an access to another checking (saving) account.

```
public class CheckingThread extends Thread {
  private int amount;
  private Vector Vcheck;

  public CheckingThread(int amount,Vector Vcheck) {
    this.amount = amount;
    this.Vcheck = Vcheck;
  }

  public void run {
    int account;
    Random R = new Random();
    while(true) {
      account = R.nextInt(Vcheck.capacity());
      ((Checkingaccount)V.get(account)).widthdraw(amount);
    }
  }
}


public class Savingthread extends Thread {
  private int amount;
  private Vector Vsav;

  public SavingThread(int amount, Vector Vsav) {
    this.amount = amount;
    this.Vsav = Vsav;
  }

  public void run {
    int account;
    while (true) {
      account = R.nextInt(Vsav.capacity());
      ((Savingaccount)V.get(account)).deposit(amount);
    }
  }
}
```

Figure 3.9: JAVA thread classes that withdraw money on a checking account and deposits money on a saving account.

```
Checkingthread {vars :;
  while(true){
    if(*) {
      lock(checklock);
      lock(savelock);
      unlock(savelock);
      unlock(checklock);
    } else {
      skip;
    }
  }
}
```

```
Savingthread{vars :;
  while(true){
    if (*)
      lock(savelock);
      lock(checklock);
      unlock(checklock);
      unlock(savelock);
    } else {
      skip;
    }
  }
}
```

Figure 3.10: Abstraction of the JAVA thread classes of Figure 3.9 obtained by using no predicate.

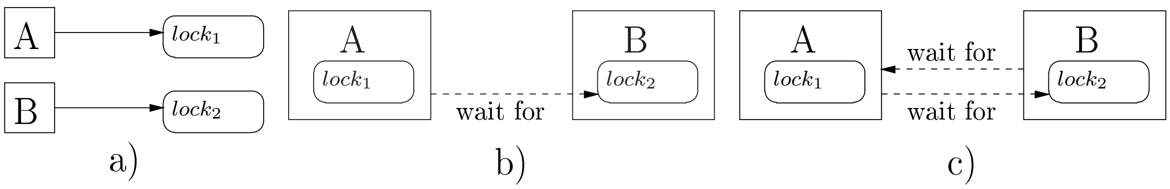

Figure 3.11: A deadlock situation

It can be easily proved (for instance by using the translation of Section 3.2 and Section 2.3 to obtain a MTN and then applying a verification algorithm presented in Chapter 5) that deadlocks occur during executions of the CBP. Moreover, executions of the JAVA program can be constructed from some of those executions of the CBP (remember that CBPs are not exact abstractions in the sense that some executions of the CBPs could not correspond to an execution of the corresponding JAVA programs). Those executions correspond to the cases where a checking account and its saving account try to make a transfer of money at the same time. In this case, the two accounts have their own lock to make a transaction (deposit or withdraw) and try to take the lock of the other one to make the transfer of money. So, we conclude that the JAVA program is not deadlock free.

Let us consider an upgraded version of the solution, proposed by D. Lea in [Lea00, pages 241-244] and shown in Figure 3.13. The solution uses TSBoolean objects (see Figure 3.12) that are shared by checking accounts and their corresponding saving account. Before trying a transfer of money, a thread calls the method testAndSet of the TSBoolean object of the account. That method decides if the transfer can be carried out safely or not. When a negative answer is returned by the method, the threads wait (using the wait method) until a positive answer is given. When a thread finishes its transfer, it invokes the clear method of the TSBoolean object and then the method awake to awake a potentially waiting thread with the notify primitive.

To avoid deadlock problems, the new version of the Checkingaccount and Savingaccount classes are used in the JAVA program instead of their bugged version. The CBP threads corresponding to this improved version of the automatic transfer service are shown in Figure

```
public class TSBoolean {

private boolean value = false;

public synchronized boolean testAndSet(){
  boolean oldvalue = value;
  value = true;
  return oldvalue;
}
public synchronized void clear(){
  value = false;
}
public synchronized void awake(){
  notify();
}
public synchronized void sleep(){
  wait();
}
}
```

Figure 3.12: A Java class for TSBoolean objects.

3.14 (the parts corresponding to accesses to objects different from O have been omitted). Those CBP threads have been obtained by using the principle of Section 3.3.1 and the variables oldval and val corresponding, respectively, to the constraints oldvalue == true and value == true on the variables value and oldvalue [5] of the TSBoolean class of the JAVA program. When applying the verification algorithm presented in Chapter 5 (after translating the CBP into a MTN as shown previously), we can automatically verify that no deadlock situation occurs in the CBP. As all the executions of the corresponding JAVA program corresponds to an execution of the CBP, we can directly conclude that the JAVA program is deadlock free.

Example 2: The distributed transaction system example

The example considered in this section is related to problems occurring in data bases when users can concurrently access to data's. In those cases, some techniques must ensure the unicity of transactions. A transaction is a sequence of operations on data's (contained in this case into JAVA objects). The unicity of a transaction means that the effect of the whole set of operations of the transaction is visible or none of them is visible. A classical technique is to use locks for each object in the database. To access an object, a transaction must first get its lock. If the lock is not available, the transaction waits until it is released. A transaction releases the locks it gets at its completion or its aborting. Obviously, some deadlock can occur if nothing is done to avoid them.

In the following, we will focus on the *unicity property* verification, i.e. in verifying that an object cannot be accessed by two different active transactions at the same time. A solution is

[5] Notice that since the methods are inlined, the oldvalue variable becomes a local variable of the run method of threads.

```
public class Checkingaccount {
protected Savingaccount saving;
protected long threshold, balance;
protected TSBoolean B = new TSBoolean();
public Checkingaccount(long t) {
  threshold=t;
  balance=0;}
synchronized void
    initSaving(Savingaccount s){
  saving=s;}
protected boolean shouldTry(){
  return balance<threshold;}
void tryTransfer(){
  while(B.testAndSet()) B.sleep();
  synchronized(this){
  if(shouldTry())
    balance+=saving.transferOut();}
  B.clear();
  B.awake();}
public synchronized boolean
            deposit(long amount){
  if(balance + amount < 0)
    return false;
  else{
    balance+=amount;
    tryTransfer();
    return true;}
}
public boolean withdraw(long amount){
  return deposit (-amount);}
}
```

```
public class Savingaccount{
protected chechingaccount checking;
protected long maxtransfer, balance;

public Savingaccount(long max){
  maxtransfer=max;
  balance = 0;}
synchronized void
    initChecking(Checkingaccount s){
  checking=c;}
synchronized long transferOut(){
  long amount=balance;
  if(amount>maxtransfer)
    amount=maxtransfer;
  if(amount>=0)
    balance-=amount;
  return amount;}
public synchronized boolean
      deposit(long amount){
  if(balance + amount < 0)
    return false;
  else {
    balance+=amount;
    checking.tryTransfer();
    return true;}
}
public boolean withdraw(long amount)
  {return deposit (-amount);}
}
```

Figure 3.13: The new JAVAclasses Checkingaccount and Savingaccount.

```
Check { vars: ;
  while (true){
    lock(checklock);
    if (*){
      goto endcheck;}
    else{
      lock(tsboollock);
      if(val)
        oldval := true;}
      else
        oldval := false;}
      val := true;
      unlock(tsboollock);
      while(oldval){
        lock(tsboollock);
        sleep(msgtsbool,tsboollock);
        unlock(tsboollock);
        lock(tsboollock);
        if(val)
          oldval := true;}
        else
          oldval := false;}
        val := true;
        unlock(tsboollock);}
      if (*) {
        lock(savelock);
        unlock(savelock);}
      else{
        skip;}
      lock(tsboollock);
      val := false;
      unlock(tsboollock);
      wakeup(msgcheck,checklock);
    }
    [endcheck]unlock(checklock);
  }
}
```

```
Save{ vars: ;
  while (true){
    lock(savelock);
    if(*) {
      goto endsave;}
    else{
      lock(tsboollock);
      if(val){
        oldval := true;}
      else{
        oldval := false;}
      val := true;
      unlock(tsboollock);
      while(oldval){
        lock(tsboollock);
        sleep(msgtsbool,tsboollock);
        unlock(tsboollock);
        lock(tsboollock);
        if(val){
          oldval := true;}
        else{
          oldval := false;}
        val := true;
        unlock(tsboollock);}
      lock(checklock);
      if (*) {
        skip;}
      else{
        skip;}
      unlock(checklock);
      lock(tsboollock);
      val := false;
      unlock(tsboollock);
      wakeup(msgcheck,checklock);
    }
    [endsav]unlock(savelock);
  }
 }
```

Figure 3.14: Abstraction of the JAVA threads of Figure 3.9 using the upgraded version of Checkingaccount and Savingaccount classes and obtained using the predicates defined by oldvalue == true (variable oldval) and value == true (variable val).

given in [Lea00, pages 252-260] (see the code of Figure 1 in Appendix which is inspired from this solution). As transactions do not release locks of objects after accessing their data's but when they terminate, the locks on objects implemented by the JAVA language are not used to ensure the unicity property. Instead, a special data in each object is used to implement the locks. More precisely, each object accessed in the system by a transaction has a reference, called lockref to the transaction that has its lock. In the code given in Figure 1 in Appendix, transactions are modeled by threads. To take the lock, a transaction only has to put its reference into lockref with an operation of the form $\mathsf{lockref} = \mathsf{t}$ (see Figure 1 in Appendix). Release the lock corresponds to operations of the form $\mathsf{lockref} = \mathsf{NULL}$.

The program that we consider is composed of several threads making one transaction (see Figure 2 of Appendix). Some instances of the class SimpleTransBankAccount are available through a global vector V. The transactions consist in accessing non deterministically some objects before completing the transaction or aborting it.

The global objects are abstracted in the same way than in the *account manager* example by keeping track of the information on only one object O chosen without assumption.

To verify the unicity property, we need to first verify that a thread A cannot acquire the lock of an object if it is not free (corresponding to verify that the operations of the form $\mathsf{lockref} = \mathsf{t}$ are never executed when $\mathsf{lockref} \neq null$), verify that a thread A cannot release a lock that it does not own (corresponding to verify that the operations of the form $\mathsf{lockref} = \mathsf{null}$ are never executed when $\mathsf{lockref} \neq A$) and then verify that a thread A cannot access the data's of O if it has not its lock (corresponding to verify that a thread cannot be at some point of the code if it has not the lock of O). To make such a verification, we need some information in the abstract model to keep track of the current thread that has the lock of O. For this reason, we abstract threads in several ways.

First, we will have two predicates $\mathsf{lockref} == A$ and $\mathsf{lockref} == null$ corresponding to two global variables of CBP lockref_eq_A and lockref_eq_null. More precisely, the variable lockref_eq_A will be equal to true when lockref contains the reference of the thread A, otherwise it will be equal to false. The variable lockref_eq_null will be equal to true when lockref is equal to $null$, otherwise it will be equal to false.

Then, we abstract the thread A, chosen without assumption, as follows. The JAVA operations of the form

$$\mathsf{lockref} = \mathsf{t};$$

are abstracted in the CBP by operations of the form

$$\mathsf{lockref\_eq\_A}, \mathsf{lockref\_eq\_null} := \mathsf{true}, \mathsf{false};$$

meaning that the thread A puts its reference into lockref. The tests of the form

$$(i)\ \mathsf{if}\ (\mathsf{lockref}\ ==\ \mathsf{t})\ \{\ S\ \}\ \text{and}\ (ii)\ \mathsf{if}\ (\mathsf{lockref}\ !=\ \mathsf{t})\ \{\ S\ \}$$

are abstracted, respectively, by tests of the form

$$(i)\ \mathsf{if}\ (\mathsf{lock\_eq\_A})\ \{\ abs(S)\ \}\ \mathsf{else}\ \{\ \mathsf{skip};\ \}\ \text{and}\ (ii)\ \mathsf{if}\ (!\mathsf{lock\_eq\_A})\ \{\ abs(S)\ \}\ \mathsf{else}\ \{\ \mathsf{skip};\ \}$$

where $abs(S)$ corresponds to the abstraction of the block S.

The other threads are abstracted losing the information related to the fact that lockref could contain their reference. This means that the JAVA operations of the form

$$\mathsf{lockref} = \mathsf{t};$$

are abstracted in the CBP by operations of the form

$$\mathsf{lockref\_eq\_A}, \mathsf{lockref\_eq\_null} := \mathsf{false}, \mathsf{false};,$$

and the tests of the form

$$(i)\ \mathsf{if}\ (\mathsf{lockref} == \mathsf{t})\ \{\ S\ \}\ \text{and}\ (ii)\ \mathsf{if}\ (\mathsf{lockref}! = \mathsf{t})\ \{\ S\ \}$$

are replaced, respectively, by

$$(i)\ \mathsf{if}\ (!\mathsf{lockref\_eq\_A}\ \mathsf{and}\ !\mathsf{lockref\_eq\_null})\ \{\mathsf{if}(*)\{\ abs(S)\ \}\}$$

and

$$(ii)\ \mathsf{if}\ (\mathsf{lockref\_eq\_A}\ \mathsf{or}\ \mathsf{lockref\_eq\_null})\ \{\ abs(S)\ \}\ \mathsf{else}\{\mathsf{if}(*)\{\ abs(S)\ \}\}$$

where $abs(S)$ corresponds to the abstraction of the block S. Notice that we have forgotten the else branches that only contain a skip; operation to make the translation more clear. The imbricated if's (allowing non-deterministic choices) correspond, in both tests (i) and (ii), to the cases where it is not possible to evaluate the test with the values of the predicates. For instance, let us consider the test (i). When !lockref_eq_A and !lockref_eq_null is evaluated to true, lockref can be possibly equal to the reference of the thread or not. So the abstract model non-deterministically decides if the concrete test (lockref== t) is evaluated to true or not.

Finally, for all the threads, the operations of the form

$$\mathsf{lockref} = \mathsf{null}$$

are abstracted into the operation

$$\mathsf{lockref\_eq\_A}, \mathsf{lockref\_eq\_null} := \mathsf{false}, \mathsf{true};$$

and the tests of the form

$$\mathsf{while}\ (\mathsf{lockref}! = \mathsf{null})\ \{\ S\ \}$$

are replaced by

$$\mathsf{while}\ (!\mathsf{lockref\_eq\_null})\ \{\ abs(S)\ \}.$$

Figure 3 in Appendix shows the CBP thread corresponding to the JAVA thread A and Figure 4 in Appendix shows the CBP thread corresponding to the other JAVA threads. The calls to the methods balance, deposit and withdraw, and the calls to the methods abort and commit are abstracted in the same way. Hence, only the abstractions of the methods balance and abort are shown in Figure 3 and Figure 4 in Appendix.

As the thread A and the global object O are chosen without assumption, this abstraction method allows us to verify the desired property on the concrete JAVA program.

This is achieved by verifying on the abstract model (i) that the abstract thread A never computes the operation lockref_eq_A, lockref_eq_null := true, false when lockref_eq_null and lockref_eq_A are both equal to false; and (ii) that the abstract thread A cannot be at some point of its execution corresponding to an access to the data of O if lockref_eq_A == false.

Those properties have been verified during the experiments reported in Section 5.5.5.

Example 3: The bounded buffer example

The bounded buffer example consists into a buffer shared by several threads. The buffer has a fixed number of slots and these slots are accessed in a circular manner. This means that the first slot is used to write some information, then the second slot is used, etc. When a thread tries to read into the buffer, it get access to the oldest written slot (which becomes free after reading). When the last slot of the buffer is used, a writing thread uses the first one again if it is free, etc. This technique allows us to simulate an unbounded buffer. However, the (implemented) buffer can become full if no thread reads into the buffer. In this case, a new writing action must be postponed until a slot becomes available. In a similar manner, a reading action is postponed when no slot contains information.

An implementation of such a buffer is proposed by D. Lea in [Lea00, pages 270-272]. This implementation is shown in Appendix, Figure 5 and Figure 6.

The buffer is defined by the class Boundedbuffer. The buffer itself is an array of references to objects that are put and get by threads.

Instantiations of this class do not access directly to the array corresponding to the buffer. Actually, there is an object putter used to put some information (an object) into the buffer and an object taker used to take some information (an object) from the buffer. The putter and taker objects are instances of the same class Exchange, defined inside the class Boundedbuffer, and hence work similarly. The putter (taker) object uses a counter ptr that contains the index of the next slot to put an object in (get an object from) and a counter slot that contains the number of free slots (the number of slots containing an object).

This special design corresponds to a delegate pattern. This pattern proposes to use a particular object to make a particular operation on a data structure. This leads to less synchronizations and less communications between threads, i.e. the goal of the pattern is to increase the performances of the system.

More precisely, threads that want to put some information into the buffer do not have to wait for a lock (to enter into a synchronized method) when a thread reads into the buffer. Moreover, writing threads and reading threads wait on two different objects. That allows us to awake writing threads only when a slot becomes available and to awake reading threads only when some slots contain information (instead of awake all the reading and writing threads).

In the following, we fix the number of slots to 3 and consider a JAVA program composed of several threads that are instances of the classes shown in Appendix, Figure 7. As in the previous examples, the main method of the JAVA program instantiates a (non-deterministically chosen) number of threads sharing a Boundedbuffer object. Figure 8 and Figure 9 in Appendix show the CBP threads corresponding to the JAVA threads (Figure 7 in Appendix). The construction of those CBP threads is done using boolean variables corresponding to the constraints

- $\mathsf{ptr_t} == \mathsf{i}$ and $\mathsf{ptr_p} == \mathsf{i}$ where $0 \leq i \leq 2$ is one of the possible indices and $\mathsf{ptr_t}$, respectively $\mathsf{ptr_p}$, refers the variable ptr of the taker object, respectively the putter object. In Figure 8 and Figure 9 of Appendix, the variable Tptri (Pptri) with $0 \leq i \leq 2$ corresponds to the constraint $\mathsf{ptr_t} == \mathsf{i}$ ($\mathsf{ptr_p} == \mathsf{i}$);

- $\mathsf{slot_t} == \mathsf{i}$ and $\mathsf{slot_p} == \mathsf{i}$ where $0 \leq i \leq 3$ and $\mathsf{slot_t}$, respectively $\mathsf{slot_p}$, refers the variable slot of the taker object, respectively the putter object. In Figure 8 and Figure 9 of Appendix, the variable Tsloti (Psloti) with $0 \leq i \leq 3$ corresponds to the constraint $\mathsf{slot_t} == \mathsf{i}$ ($\mathsf{slot_p} == \mathsf{i}$).

Several properties can be automatically checked on the resulting CBP. An important property is the exclusive access of a slot, i.e. a slot cannot be accessed at the same time by a reading thread and a writing thread. This can be checked for the slot l ($0 \leq l \leq 2$) by verifying that it is not possible to reach a state where the variables $\mathsf{ptr_t} == \mathsf{l}$, $\mathsf{ptr_p} == \mathsf{l}$, $\mathsf{slot_p} == \mathsf{i}$ and $\mathsf{slot_t} == \mathsf{j}$ for some $i, j \geq 1$ are evaluated to true. Two other interesting properties to verify are that *a slot accessed to take information is never empty* and that *a slot accessed to put information is always empty*. This can be done for the slot i by refining the model, i.e. by adding a variable giving the value of the constraint $\mathsf{buffer[i]} == \mathsf{NULL}$. Then, we verify the safety of the JAVA program by verifying that the CBP cannot reach a configuration where $\mathsf{ptr_t} == \mathsf{i}$ ($\mathsf{ptr_p} == \mathsf{i}$) is true, $\mathsf{buffer[i]} == \mathsf{NULL}$ is also true (false) and there is a thread that takes (puts) information into the buffer.

Those properties have been verified during the experiments reported in Section 5.5.5.

3.3.3 A general approach for the verification of JAVA programs

In the field of verifying safety properties, a challenging problem with predicate abstraction is to find a good compromise between the precision of the abstract models and their size i.e. the number of predicates used for the construction of the abstract model, and the size of the abstract model that will be analyzed. A second problem is that although predicate abstraction is an adequate abstraction technique, finding the *good* predicates for the verification of real systems is a very time consuming operation and needs a good knowledge of the concrete system (see [DDP99]). By good predicates, we mean predicate that will allow us to construct an abstract model from which some information about the real system can be obtained.

The technique investigated in [SS99] to deal with the first problem consists to start with a poor abstract model constructed using a small set of predicates S (it is not necessary to start with the set of predicates corresponding to tests in the JAVA programs), and then refine it. When we detect that the model is too coarse to verify the desired properties, a set of new predicates S' is constructed and a refined abstract model is constructed using the predicates in $S \cup S'$. In [BPR02, HJMS02], this strategy is applied to sequential software verification. Finite automata are automatically constructed from sequential c programs using predicate abstraction. The problem of finding good predicates is avoided by (automatically) analyzing false error executions. False error executions are executions of the abstract model that do not satisfy the questioned property but do not correspond to an execution of the concrete program.

Analyzing coarse abstract models often leads to a false negative answer of the model checker. Typically, these executions appear because of the non determinism introduced by the predicate abstraction. The non-determinism appears when their is not enough information in the abstract model to mimic exactly an operation or a test of the real system. For instance, if the abstract model only knows if a variable x is positive or negative, then testing if x is equal to 5 cannot be tested exactly. Indeed, in the case where the abstract model considers that x is positive, it cannot decide if x is equal to 5 or equal to another positive value, and so it will have the possibility to non deterministically consider that the test is positive or negative.

False negative answers of the model checker is a direct consequence of the fact that the abstract model can make wrong choices during its executions. In our example, the abstract model could choose to consider x equal to 5 at some point where x is never equal to 5 in the real system leading to the detection of errors in the abstract model that does not exist in the real system.

Analyzing the false error executions, bad choices are (automatically) detected leading to add some new predicates avoiding these choices, i.e. the use of the new predicates removes the detected false error execution (and potentially other ones) from the set of executions of the abstract model. The refined model is then analyzed. The refinement process ends in three cases:

1. the abstract model is proved to be safe. The safety property of the original program is then directly deduced;
2. an error execution of the abstract model is detected and allows us to construct a corresponding execution of the original program where some mistakes occur;
3. the verification process is aborted.

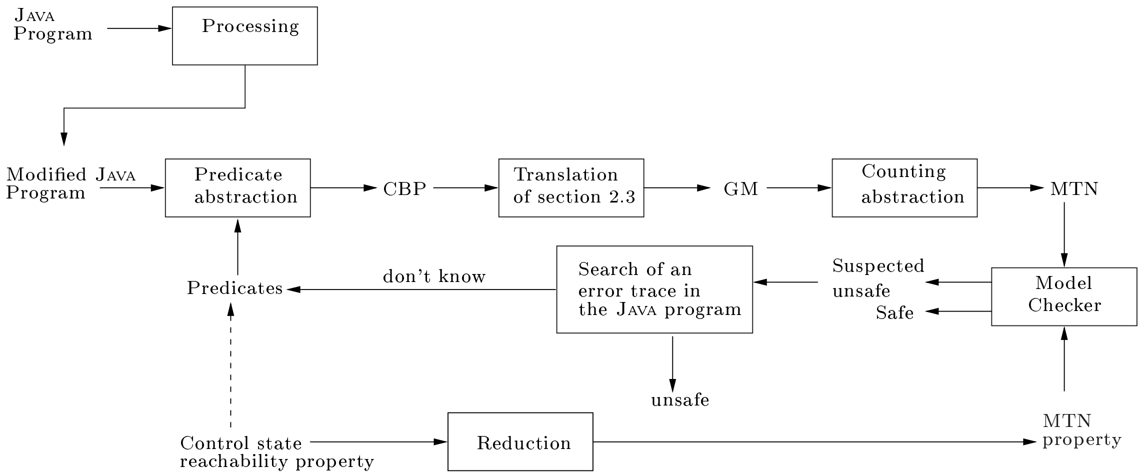

Figure 3.15: The verification process of JAVA programs

Although our approach consists in constructing infinite state models (that can be finitely described), we sometimes construct too coarser models that must be refined as in [BPR02, HJMS02]. We propose to use a similar approach than in the finite state case.

Figure 3.15 summarizes the different phases of the verification process of JAVA programs. First, the JAVA program to verify is slightly modified, for instance by inlining the method calls. The result is a JAVA program that is then abstracted by applying the predicate abstraction with an initial set of predicates S. This abstraction allows us to verify programs with data's having an infinite domain. This phase leads to a CBP which is translated into a family of global machines using the translation of section 2.3. Then, the counting abstraction is applied on the family of global machines to obtain a MTN. This abstraction allows us to verify programs with an unbounded number of threads.

The given property to verify is a control state reachability property that should be verified by the CBP. Remember that the control state reachability problem consists to decide if there is a reachable global state of the CBP where the global variables satisfy a constraint and there is minimal number of thread instances that reached some labels. We suppose here that the constraint on the global variables of CBPs relates to variables corresponding to predicates given in the initial set of predicates, i.e. the constraint relates to variables that will appear in all the CBPs that are constructed during the refinement process.

The property is reduced to a covering property on MTN. Finally, the MTN and the property on the MTN is given to a model checker. Several answers can be returned by the model checker: either the MTN satisfies the property to verify or not.In the first case, the verification process finishes and a positive answer is returned to the user. In the other case, if an error execution of the Java program can be constructed, the process also terminates and a negative answer is returned to the user. Otherwise, new predicates are added into S and the verification process is started again.

As in the case of the sequential programs verification, this process may loop forever. This is a direct consequence of the undecidability of all the non trivial verification problems on programs.

In the next chapters, we focus on algorithmic problems encountered when implementing an efficient model checker. More precisely, we will present efficient data structures to represent special classes of sets and verification algorithms using them that allow us to verify covering problems on MTNs corresponding to abstractions of Java programs.

Chapter 4

Handling Closed Sets

In this chapter, we study two special kinds of infinite sets of tuples called *upward closed sets* and *downward closed sets* of tuples over integers. This study is motivated by the fact that those classes are manipulated by the verification algorithms on MTNs presented in Chapter 5.

First, we define two logics, called $\mathcal{U}$-logic and $\mathcal{D}$-logic, that describe exactly those two classes of sets. Then, we study the complexity of classical operations on formulas of these logics.

In the second part of the chapter, we present a symbolic automata-based data structure, called *Covering Sharing Tree*, allowing to efficiently encode formulas of the $\mathcal{U}$-logic and the $\mathcal{D}$-logic. We also study the complexity of symbolic operations on the structure corresponding to operations on formulas.

We finish the chapter by some considerations on heuristics based on simulation relation computations.

The content of this chapter is based on the paper [DRB02].

4.1 Preliminaries

Let D and E be two disjoint sets such that $D \cap E = \emptyset$. We call the elements of E the limit elements. Let $\sqsubseteq$ be a partial order on $D \cup E$. We say that e subsumes e' when $e \sqsubseteq e'$. When neither $e \sqsubseteq e'$ nor $e' \sqsubseteq e$, we say that e and e' are incomparable. We note $e \sqsubset e'$ when we have $e \sqsubseteq e'$ with $e' \neq e$. In the following, we note $(D, E, \sqsubseteq)$ for the domain D and the set of limit elements E with the partial order $\sqsubseteq$ on $D \cup E$. Sometimes we omit E when we only consider elements in D [1]. A *closed set* in $(D, \sqsubseteq)$ is a set $S \subseteq D$ such that for all $e \in S$ if $e \sqsubseteq e'$, then $e' \in S$. A *generator set* of a closed set S in $(D, \sqsubseteq)$ is a set $L \subseteq (D \cup E)$ such that $S = \{e \in D \mid \exists e' \in L : e' \sqsubseteq e\}$. A *minimal generator set* of S is a generator set L of S such that for all $e \in L$ the set $L \setminus \{e\}$ is not a generator set of S. We denote by $\mathit{expand}(e)$ the set generated by the element $e \in D \cup E$, i.e. $\mathit{expand}(e) = \{e' \in D \mid e \sqsubseteq e'\}$. $\mathit{expand}(.)$ is extended naturally to (generator) sets $L \subseteq (D \cup E)$.

Figure 4.2 shows an example where the set D contains all the rationals 2^{-n} with $n \in \mathbb{N}$, $E = \{0\}$ and the partial order is $\leq$. If we consider the set $\{1/16\}$ as a generator set, then it generates the set $\{1, 1/2, 1/4, 1/8, 1/16\}$. In this example, no element in D allows us to

[1] Notice that, as $\sqsubseteq$ is a partial order on the elements in $D \cup E$, $\sqsubseteq$ is a partial order on the elements in D.

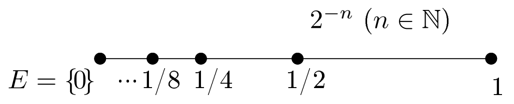

Figure 4.1: An example where limit elements are useful.

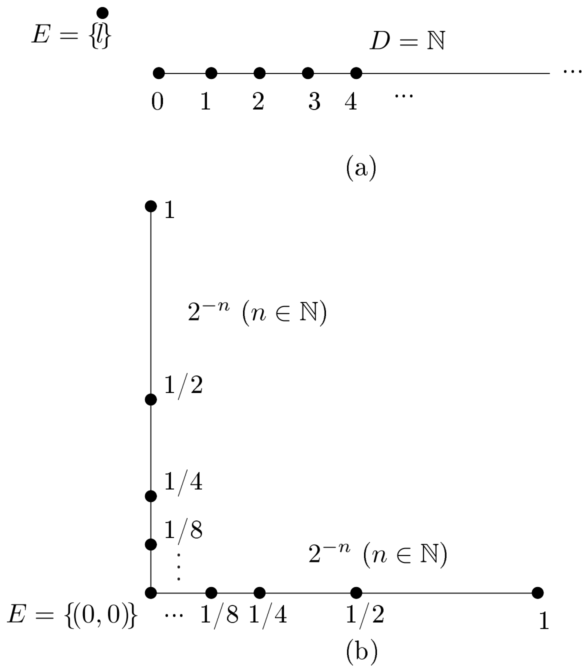

Figure 4.2: (a) an example where the partial order does not satisfy the property (i) of a well-suited partial order, (b) an example where the partial order does not satisfy the property (ii) of a well-suited partial order.

generate the whole set D. Actually, all the elements in D have elements that are strictly smaller. To finitely represent D with a generator set, we need to use the limit element 0.

We say that $\sqsubseteq$ is *well-suited* for $(D, E, \sqsubseteq)$ if (i) for all $e, e' \in D \cup E$, if $\mathit{expand}(e') \subseteq \mathit{expand}(e)$ then $e \sqsubseteq e'$, and (ii) for all $e \in E$ and for all $e', e'' \in D$ such that $e \sqsubseteq e'$ and $e \sqsubseteq e''$, there exists $e''' \in D$ with $e \sqsubseteq e'''$ such that $e''' \sqsubseteq e'$ and $e''' \sqsubseteq e''$. The first condition means that the partial order on the limit elements reflects the (inverse of the) inclusion relation on the sets that they generate. The second condition says that when two elements in D are generated by an element e in E, there is a third element in D generated by e that generates them. Notice that property (ii) is trivially true for $e \in D$ (one may simple take $e''' = e$).

Figure 4.2(a) shows an example where $D = \mathbb{N}$, $E = \{l\}$ and the partial order is the total order over natural numbers extended to l such that $l \not\leq i$ and $i \not\leq l$ for all $i \in \mathbb{N}$. In this case, $\leq$ is not well-suited because l generates the empty set, i.e. $\mathit{expand}(l) \subseteq \mathit{expand}(i)$ for all $i \in \mathbb{N}$, and $i \not\leq l$ for all $i \in \mathbb{N}$. Figure 4.2(b) shows an example where $D = \{(k, 0)$ such that $k = 2^{-i}$ for some $i \geq 0\} \cup \{(0, k)$ such that $k = 2^{-i}$ for some $i \geq 0\}$, $E = \{(0, 0)\}$ and the order $\sqsubseteq$ is defined as follows: $(n, m) \sqsubseteq (n', m')$ if and only if $n \leq n'$ and $m \leq m'$. In this example, $\sqsubseteq$ is not well-suited because the property (ii) is not verified. For instance, for the tuples $(0, 1/2)$

and $(1/2, 0)$ there is no tuple $(n, m) \in D$ such that $(n, m) \sqsubseteq (0, 1/2)$ and $(n, m) \sqsubseteq (1/2, 0)$. Figure 4.1 shows an example where the partial order is well suited (see the explanation about the example above).

$(D, E, \sqsubseteq)$ is *well-formed* if $\sqsubseteq$ is well-suited, and if every closed set in $(D, \sqsubseteq)$ has a finite generator set $L \subseteq D \cup E$.

An interesting property of closed sets on a well formed domain is that they have a *unique finite* minimal generator set. Before proving this result, we prove the following proposition.

Proposition 3 Let $(D, E, \sqsubseteq)$ such that $\sqsubseteq$ is well-suited and S a closed set in $(D, \sqsubseteq)$. If S has a minimal generator set, then it is unique.

Proof. Suppose that it is not the case. There exists $L, L' \subseteq D \cup E$ with $L \neq L'$ such that L and L' are minimal generator sets of S. Without loss of generality, suppose that $e \in L$ but $e \notin L'$. We consider the following cases.

- There is $e' \in L'$ such that $\mathit{expand}(e) \subseteq \mathit{expand}(e')$. As $e \neq e'$ by hypothesis and $\sqsubseteq$ is well-suited, we have that $\mathit{expand}(e) \subset \mathit{expand}(e')$. Indeed, if $\mathit{expand}(e) = \mathit{expand}(e')$, by property (i) of a well-suited partial order, we have $e \sqsubseteq e'$ and $e' \sqsubseteq e$. Since $\sqsubseteq$ is a partial order, we obtain $e = e'$. That makes a contradiction with the fact that $e \neq e'$ and we conclude that we have $\mathit{expand}(e) \subset \mathit{expand}(e')$. Hence, there exists $s' \in \mathit{expand}(e')$ but $s' \notin \mathit{expand}(e)$. For all $s \in \mathit{expand}(e)$, $s \in \mathit{expand}(e')$ and, from property (ii) of a well-suited partial order, there exists $s'' \in \mathit{expand}(e')$ with $s'' \sqsubseteq s'$ and $s'' \sqsubseteq s$. Moreover, $s'' \notin \mathit{expand}(e)$. Indeed, suppose that it is not the case and we have $s'' \in \mathit{expand}(e)$. As $s'' \sqsubseteq s'$, we should have $s' \in \mathit{expand}(e)$ but this makes a contradiction with the fact that $s' \notin \mathit{expand}(e)$. So we have that $s'' \neq s$ and hence $s'' \sqsubset s$. Moreover, as $S = \mathit{expand}(L) = \mathit{expand}(L')$, we have that there exists $e'' \in L$ (with $e'' \neq e$) such that $s'' \in \mathit{expand}(e'')$ and hence $s \in \mathit{expand}(e'')$.

  We conclude that for all $s \in \mathit{expand}(e)$, there exists $e'' \in L$ with $e'' \neq e$ such that $s \in \mathit{expand}(e'')$. Hence, $\mathit{expand}(L) = \mathit{expand}(L \setminus \{e\})$. This makes a contradiction with the fact that L is minimal.

- There is $e' \in L'$ such that $\mathit{expand}(e') \subseteq \mathit{expand}(e)$. This case is similar to the preceding one. As $e \neq e'$ by hypothesis and $\sqsubseteq$ is well-suited, we have that $\mathit{expand}(e') \subset \mathit{expand}(e)$. Indeed, if $\mathit{expand}(e') = \mathit{expand}(e)$, by property (i) of a well-suited partial order, we have $e \sqsubseteq e'$ and $e' \sqsubseteq e$. Since $\sqsubseteq$ is a partial order, we obtain $e = e'$. That makes a contradiction with the fact that $e \neq e'$ and we conclude that we have $\mathit{expand}(e') \subset \mathit{expand}(e)$. Hence, there exists $s \in \mathit{expand}(e)$ but $s \notin \mathit{expand}(e')$. For all $s' \in \mathit{expand}(e')$, $s' \in \mathit{expand}(e)$ and, from property (ii) of a well-suited partial order, there exists $s'' \in \mathit{expand}(e)$ with $s'' \sqsubseteq s'$ and $s'' \sqsubseteq s$. Moreover, $s'' \notin \mathit{expand}(e')$. Indeed, suppose that it is not the case and we have $s'' \in \mathit{expand}(e')$. As $s'' \sqsubseteq s$, we should have $s \in \mathit{expand}(e')$ but this makes a contradiction with the fact that $s \notin \mathit{expand}(e')$. So we have that $s'' \neq s'$ and hence $s'' \sqsubset s'$. Moreover, as $S = \mathit{expand}(L) = \mathit{expand}(L')$, we have that there exists $e'' \in L'$ with $e'' \neq e'$ such that $s'' \in \mathit{expand}(e'')$ and hence $s' \in \mathit{expand}(e'')$.

  We conclude that for all $s' \in \mathit{expand}(e')$, there exists $e'' \in L'$ with $e'' \neq e'$ such that $s' \in \mathit{expand}(e'')$. Hence, $\mathit{expand}(L') = \mathit{expand}(L' \setminus \{e'\})$. This makes a contradiction with the fact that L' is minimal.

- for all $e' \in \mathit{expand}(L')$ we have $\mathit{expand}(e) \not\subseteq \mathit{expand}(e')$ and $\mathit{expand}(e) \not\subseteq \mathit{expand}(e')$. As $S = \mathit{expand}(L) = \mathit{expand}(L')$, for all $s \in \mathit{expand}(e)$ there is $e' \in L'$ such that

$s \in \mathit{expand}(e')$. Moreover, as $\mathit{expand}(e') \not\subseteq \mathit{expand}(e)$, there exists $s' \in \mathit{expand}(e')$ with $s' \notin \mathit{expand}(e)$. Following the property (*ii*) of a well-suited partial orderings, there exists $s'' \in \mathit{expand}(e')$ such that $s'' \sqsubseteq s'$ and $s'' \sqsubseteq s$. Moreover, $s'' \notin \mathit{expand}(e)$. Indeed, if we have $s'' \in \mathit{expand}(e)$, as $s'' \sqsubseteq s'$, we would have $s' \in \mathit{expand}(e)$. But this makes a contradiction. Hence, $s'' \sqsubset s$.

Moreover, as $\mathit{expand}(L) = \mathit{expand}(L')$, we have that there exists $e'' \in L$ with $e'' \neq e$ such that $s'' \in \mathit{expand}(e'')$ and hence $s \in \mathit{expand}(e'')$.

We conclude that for all $s \in \mathit{expand}(e)$, there exists $e'' \in L$ with $e'' \neq e$ such that $s \in \mathit{expand}(e'')$. Hence, $\mathit{expand}(L) = \mathit{expand}(L \setminus \{e\})$. This makes a contradiction with the fact that L is minimal.

In the three cases, we obtain a contradiction. We conclude that L does not generate the same closed set than L', i.e. $\mathit{expand}(L) \neq \mathit{expand}(L')$. Thus, if L is a minimal generator set of the closed set S, it is unique. □

From Proposition 3, we have the following corollary.

Corollary 4 *Let $(D, E, \sqsubseteq)$ be well-formed and S be a closed set in $(D, \sqsubseteq)$. Then, S has an unique finite minimal generator set.*

Proof. As $(D, E, \sqsubseteq)$ is well-formed, S has a finite generator set L. As L is finite, we can construct a finite maximal sequence $L_1, L_2, \ldots, L_n$ such that (*i*) $L_1 = L$, and (*ii*) $L_i = L_{i-1} \setminus \{e_i\}$ with $e_i \in L_{i-1}$ and $\mathit{expand}(L_i) = \mathit{expand}(L_{i-1})$. As that sequence is maximal, there is no $e \in L_n$ such that $\mathit{expand}(L_n) = \mathit{expand}(L_n \setminus \{e\})$, hence L_n is a minimal finite generator set of S and from Proposition 3 we have that it is unique. □

Finally, the following proposition characterizes the elements contained into finite minimal generator sets.

Proposition 4 *Let $\sqsubseteq$ be well-suited for $(D, E, \sqsubseteq)$. If L is a finite generator set of S such that all the elements of L are pairwise incomparable with respect to $\sqsubseteq$, then L is minimal.*

Proof. Suppose that it is not the case and there exists $e \in L$ such that $\mathit{expand}(L) = \mathit{expand}(L \setminus \{e\})$. Suppose that $L \setminus \{e\} = \{e_1, \ldots, e_n\}$. We construct in n steps an element $x \in \mathit{expand}(e)$ such that for each e_i $(1 \leq i \leq n)$ we have $x \notin \mathit{expand}(e_i)$. At the *ith* step, we construct x_i such that $x_i \in \mathit{expand}(e)$ and $x_i \notin \mathit{expand}(e_j)$ for all j such that $1 \leq j \leq i$.

More precisely, at the first step, we define x_1 as follows. Since $e_1 \not\sqsubseteq e$, by property (*i*) of a well-suited partial order, there exists at least one $x \in \mathit{expand}(e)$ such that $x \notin \mathit{expand}(e_1)$. Let x_1 be such an element.

At the *ith* step, assuming that we have constructed at the $(i-1)th$ step x_{i-1} such that $x_{i-1} \in \mathit{expand}(e)$ and $x_{i-1} \notin \mathit{expand}(e_j)$ for all j such that $1 \leq j \leq i-1$, we construct x_i as follows. As $e_i \not\sqsubseteq e$, by property (*i*) of a well-suited partial order, there exists at least one $x \in \mathit{expand}(e)$ such that $x \notin \mathit{expand}(e_i)$. Since $x_{i-1} \in \mathit{expand}(e)$, by property (*ii*) of a well-suited partial order, there exists x' such that $x' \in \mathit{expand}(e)$, $x' \sqsubseteq x_{i-1}$ and $x' \sqsubseteq x$. We define x_i such that $x_i = x'$. Since (*i*) $x_{i-1} \notin \mathit{expand}(e_j)$ for all j such that $1 \leq j \leq i-1$ and $x_i \sqsubseteq x_{i-1}$ (hence $x_i \notin \mathit{expand}(e_j)$ for all j such that $1 \leq j \leq i-1$), and (*ii*) $x \notin \mathit{expand}(e_i)$ and $x_i \sqsubseteq x$ (hence $x_i \notin \mathit{expand}(e_i)$), we have $x_i \notin \mathit{expand}(e_j)$ for all j such that $1 \leq j \leq i$.

We conclude that at the nth step, we construct $x_n \in expand(e)$ with $x_n \notin expand(e_j)$ for all j such that $1 \leq j \leq n$ and hence $expand(L) \neq expand(L \backslash \{e\})$. That makes a contradiction. □

Closed Sets in $\mathbb{Z}^k$ and $\mathbb{N}^k$

In the rest of this section, we consider the component-wise ordering $\preccurlyeq$ defined at the beginning of Section 2.2 extended to $\mathbb{Z}^k$. More precisely, $\preccurlyeq$ over tuples in $\mathbb{Z}^k$ is defined as follows: $\mathbf{t} \preccurlyeq \mathbf{t'}$ if and only if $t_i \leq t'_i$ for $1 \leq i \leq k$.

If we associate the limit elements $+\infty$ and $-\infty$ to $\mathbb{Z}$, then the ordering on integers is naturally extended by taking $-\infty < c < +\infty$ for any $c \in \mathbb{Z}$. The component-wise ordering $\preccurlyeq$ is extended in the natural way.

Given a tuple $\mathbf{t} \in (\mathbb{Z} \cup \{-\infty\})^k$, we define $cone(\mathbf{t})$ as the *upward-closed set* (or *cone*) generated by $\mathbf{t}$, namely,

$$cone(\mathbf{t}) = \{\mathbf{t'} \in \mathbb{Z}^k \mid \mathbf{t} \preccurlyeq \mathbf{t'}\}.$$

Similarly, for a tuple $\mathbf{t} \in (\mathbb{Z} \cup \{+\infty\})^k$, we define $dcone(\mathbf{t})$ as the *downward-closed set* (or *downward cone*) generated by $\mathbf{t}$, namely,

$$dcone(\mathbf{t}) = \{\mathbf{t'} \in \mathbb{Z}^k \mid \mathbf{t'} \preccurlyeq \mathbf{t}\}.$$

We extend these definitions to a set S in the natural way. A set S of tuples is *upward-closed* (*downward-closed*) if and only if $S = cone(S)$ ($S = dcone(S)$).

When restricted to *non-negative* integer values, the ordering $\preccurlyeq$ turns out to be a *well-quasi ordering* [ACJT96, FS01], i.e., for any infinite sequence $\mathbf{t_1 t_2} \ldots$ of tuples in $\mathbb{N}^k$ there always exist two indices $i < j$ such that $\mathbf{t_i} \preccurlyeq \mathbf{t_j}$. We now prove that $\preccurlyeq$ on $\mathbb{N}^k$ allows us to finitely and uniquely represent upward and downward closed set of tuples over natural numbers.

Let $E^k_{\mathbb{Z}} = \{\mathbf{t} \in (\mathbb{Z} \cup \{-\infty\})^k \mid$ there is at least one $1 \leq i \leq k$ with $t_i = -\infty\}$

The following proposition will be useful to prove that upward closed sets can be uniquely described using generator sets.

Proposition 5 *$\preccurlyeq$ is well-suited for $(\mathbb{Z}^k, E^k_{\mathbb{Z}}, \preccurlyeq)$ and $(\mathbb{N}^k, \emptyset, \preccurlyeq)$.*

Proof. We exhibit here a proof for $(\mathbb{Z}^k, E^k_{\mathbb{Z}}, \preccurlyeq)$ but this proof can be directly adapted to $(\mathbb{N}^k, \emptyset, \preccurlyeq)$ (notice that the property (ii) of a well-formed partial order is trivially verified in this last case as there is no limit element). The property (i) is proved as follows. Suppose that there exist $\mathbf{t}_1, \mathbf{t}_2 \in \mathbb{Z}^k \cup E^k_{\mathbb{Z}}$ such that $cone(\mathbf{t}_1) \subseteq cone(\mathbf{t}_2)$ but $\mathbf{t}_2 \not\preccurlyeq \mathbf{t}_1$. There exists i such that $1 \leq i \leq k$ and $t_{2i} > t_{1i}$. If $t_{1i} \in \mathbb{N}$, then there exists $\mathbf{t} \in cone(\mathbf{t}_1)$ with $t_i = t_{1i}$ and hence $\mathbf{t} \notin cone(\mathbf{t}_2)$. Otherwise $t_{1i} = -\infty$ and hence $t_{2i} \in \mathbb{N}$ since $t_{1i} < t_{2i}$. There exists $\mathbf{t} \in cone(\mathbf{t}_1)$ with $t_i = t_{1i} - 1$ and hence $\mathbf{t} \notin cone(\mathbf{t}_2)$. In both cases, we obtain a contradiction with the fact that $cone(\mathbf{t}_1) \subseteq cone(\mathbf{t}_2)$ and we conclude that if $cone(\mathbf{t}_1) \subseteq cone(\mathbf{t}_2)$, then $\mathbf{t}_2 \preccurlyeq \mathbf{t}_1$.

We prove the property (ii) of a well-suited partial order as follows. Let $\mathbf{t} \in E^k_{\mathbb{Z}}$ and $\mathbf{t}_1, \mathbf{t}_2$ be two tuples such that $\mathbf{t}_1, \mathbf{t}_2 \in \mathbb{Z}^k$, $\mathbf{t} \preccurlyeq \mathbf{t}_1$ and $\mathbf{t} \preccurlyeq \mathbf{t}_2$. We construct $\mathbf{t}_3$ as follows. For all $1 \leq i \leq k$, we have $t_{3i} = min(t_{1i}, t_{2i})$ where $min(c_1, c_2) = c_1$ if $c_1 \leq c_2$, otherwise $min(c_1, c_2) = c_2$. By construction, we have that $\mathbf{t}_3 \in \mathbb{Z}^k$, $\mathbf{t} \preccurlyeq \mathbf{t}_3$, $\mathbf{t}_3 \preccurlyeq \mathbf{t}_1$ and $\mathbf{t}_3 \preccurlyeq \mathbf{t}_2$.

So, we conclude that $\preccurlyeq$ is well-suited. □

If we restrict our attention to upward closed set, we obtain the following proposition. As corollary, we have that any upward closed set in $\mathbb{N}^k$ has an unique finite minimal generator set.

Proposition 6 $(\mathbb{N}^k, \emptyset, \preccurlyeq)$ is well-formed.

Proof. From Proposition 5, we have that $\preccurlyeq$ is well-suited.

It remains to prove that any upward closed set $U \subseteq \mathbb{N}^k$ with respect to $\preccurlyeq$ has a finite set of generators. Suppose that it is not the case.

First, notice that as $\preccurlyeq$ is a well quasi ordering, there does not exist an infinite sequence $\mathbf{t}_1\mathbf{t}_2\ldots$ with $\mathbf{t}_i \in \mathbb{N}^k$ and $\mathbf{t}_i \succ \mathbf{t}_{i+1}$ for all $i \geq 1$. Indeed, there is no $i < j$ in such a sequence with $\mathbf{t}_i \preccurlyeq \mathbf{t}_j$. So, for any $U \subseteq \mathbb{N}^k$ and $\mathbf{t} \in U$ we can always construct a finite sequence (possibly of length 1) $\mathbf{t}_1\mathbf{t}_2\ldots\mathbf{t}_n$ with $\mathbf{t}_1 = \mathbf{t}$, $\mathbf{t}_i \in U$ for all $1 \leq i \leq n$ and $\mathbf{t}_i \succ \mathbf{t}_{i+1}$ for $1 \leq i < n$. Moreover, for all $\mathbf{t}' \in U$ with $\mathbf{t}' \neq \mathbf{t}_n$ we have $\mathbf{t}' \not\preccurlyeq \mathbf{t}_n$. We conclude that the set $S_{lim} = \{\mathbf{t} \in U \mid \text{there is no } \mathbf{t}' \in U \text{ with } \mathbf{t}' \prec \mathbf{t}\}$ is such that $expand(S_{lim}) = U$, i.e. S_{lim} is a generator set of U.

As, S_{lim} is a generator set of U and, by hypothesis, we have that S_{lim} is infinite. So, we can construct an infinite sequence $\mathbf{t}_1\mathbf{t}_2\ldots$ with $\mathbf{t}_i \in S_{lim}$ and $\mathbf{t}_i \neq \mathbf{t}_j$ for all $i \neq j \geq 1$.

As $\preccurlyeq$ is a well quasi ordering, there exist $i < j$ such that $\mathbf{t}_i \preccurlyeq \mathbf{t}_j$. But by construction, all the elements of S_{lim} are pairwise incomparable. That makes a contradiction and we conclude that any upward closed set $U \subseteq \mathbb{N}^k$ with respect to $\preccurlyeq$ has a finite generator set.

Finally, as $\preccurlyeq$ is well-suited and any upward closed set $U \subseteq \mathbb{N}^k$ has a finite generator set, we conclude that $(\mathbb{N}^k, \emptyset, \preccurlyeq)$ is well-formed. □

Corollary 5 *Any upward closed set in* $(\mathbb{N}^k, \emptyset, \preccurlyeq)$ *has an unique finite minimal generator set.*

Note that this property does not hold for $(\mathbb{Z}^k, \emptyset, \preccurlyeq)$ if $k \geq 2$. For instance the upward closed set in $\mathbb{Z}^k$ defined by the generator set $\{\langle i, 0, \ldots, 0, -i\rangle \mid i \in \mathbb{N}\}$ has no finite generator set.

It is easy to show that $\succcurlyeq$ is not a well-quasi ordering on elements in $\mathbb{N}^k$. For instance, the downward closed set $\{\langle i, 0, \ldots, 0\rangle \mid i \in \mathbb{N}\}$ has no finite generator set. However, downward closed sets of tuples in $\mathbb{N}^k$ can be finitely and uniquely described by a finite set of generator elements if we add limit elements, as stated by Corollary 6. In the following, we will always consider $+\infty$ as the limit element of $\mathbb{N}$ for $\geq$.

Before proving that downward closed sets of tuples in $\mathbb{N}^k$ can be finitely and uniquely represented by finite sets of limit elements, we prove that $\preccurlyeq$ is still a well quasi ordering when $\mathbb{N}$ is extended with $+\infty$.

Lemma 9 *The partial order* $\preccurlyeq$ *on elements of* $(\mathbb{N} \cup \{+\infty\})^k$ *is a well quasi ordering.*

Proof. We have to prove that for any infinite sequence $S = \mathbf{t}_1\mathbf{t}_2\ldots$ with $\mathbf{t}_i \in (\mathbb{N} \cup \{+\infty\})^k$ $(i \geq 1)$ there is $i < j$ such that $\mathbf{t}_i \preccurlyeq \mathbf{t}_j$.

As the dimension k of the tuples that we consider is finite, there exists an infinite subsequence $S' = \mathbf{t}_{i_1}\mathbf{t}_{i_2}\ldots$ of S and $I \subseteq \{1, \ldots, k\}$ such that for all $j \geq 1$ we have $t_{i_j l} = +\infty$ if and only if $l \in I$. If $I = \{1, \ldots, k\}$, then all the elements of S' are the tuple $(+\infty)^k$ and $\mathbf{t}_{i_j} = \mathbf{t}_{i_k}$ for all $j, k \geq 1$. We conclude that there exist $i < j$ such that $\mathbf{t}_i = \mathbf{t}_j$ and hence $\mathbf{t}_i \preccurlyeq \mathbf{t}_j$.

If $I \subset \{1, \dots, k\}$, then the following holds. To simplify the presentation, let us suppose that $I = \{m+1, m+2, \dots, k\}$. Let $\mathbf{t}'_{i_1}\mathbf{t}'_{i_2}\dots$ be the infinite sequence of tuples in $\mathbb{N}^m$ such that for all $j \geq 1$ and $1 \leq l \leq m$, $t'_{i_j l} = t_{i_j l}$.

As $\preccurlyeq$ is a well quasi ordering over tuples in $\mathbb{N}^m$, there exist $l < l'$ such that $\mathbf{t}'_{i_l} \preccurlyeq \mathbf{t}'_{i_{l'}}$. It follows that $\mathbf{t}_{i_l} \preccurlyeq \mathbf{t}_{i_{l'}}$, and as S' is a subsequence of S, we conclude that there exists $i < j$ such that $\mathbf{t}_i \preccurlyeq \mathbf{t}_j$. □

Let $\mathcal{E}^k_{\mathbb{Z}} = \{\mathbf{t} \in (\mathbb{Z} \cup \{+\infty\})^k \mid$ there is at least one $1 \leq i \leq k$ with $t_i = +\infty\}$ and $\mathcal{E}^k = \{\mathbf{t} \in (\mathbb{N} \cup \{+\infty\})^k \mid$ there is at least one $1 \leq i \leq k$ with $t_i = +\infty\}$.

The following proposition will be useful to prove that downward closed sets can be uniquely described using generator sets.

Proposition 7 *$\succcurlyeq$ is well-suited for $(\mathbb{Z}^k, \mathcal{E}^k_{\mathbb{Z}}, \succcurlyeq)$ and $(\mathbb{N}^k, \mathcal{E}^k, \succcurlyeq)$.*

Proof. The structures $(\mathbb{Z}^k, \mathcal{E}^k_{\mathbb{Z}}, \succcurlyeq)$ and $(\mathbb{Z}^k, E^k_{\mathbb{Z}}, \preccurlyeq)$ are isomorphic. Indeed, the mapping that associates to each tuple $\mathbf{t}$ the tuple $\mathbf{t}'$ such that $t'_i = -t_i$ for all $1 \leq i \leq k$ is an order isomorphism between the two structures. Following Proposition 5, we obtain that $\succcurlyeq$ is well-suited for $(\mathbb{Z}^k, \mathcal{E}^k_{\mathbb{Z}}, \succcurlyeq)$.

Let us prove that $\succcurlyeq$ is well-suited for $(\mathbb{N}^k, \mathcal{E}^k, \succcurlyeq)$. The property (i) holds for $(\mathbb{N}^k, \mathcal{E}^k, \succcurlyeq)$ since it holds for $(\mathbb{Z}^k, \mathcal{E}^k_{\mathbb{Z}}, \succcurlyeq)$. Indeed, suppose that there exist $\mathbf{t}_1, \mathbf{t}_2 \in \mathbb{N}^k \cup \mathcal{E}^k$ such that $dcone(\mathbf{t}_2) \subseteq dcone(\mathbf{t}_1)$ but $\mathbf{t}_2 \not\preccurlyeq \mathbf{t}_1$ in $(\mathbb{N}^k, E^k, \succcurlyeq)$. As $\succcurlyeq$ is well-suited for $(\mathbb{Z}^k, \mathcal{E}^k_{\mathbb{Z}}, \succcurlyeq)$ and $\mathbf{t}_2 \not\preccurlyeq \mathbf{t}_1$, following property (i) of a well-suited partial order, there exists $\mathbf{t} \in \mathbb{Z}^k$ such that $\mathbf{t} \preccurlyeq \mathbf{t}_2$ but $\mathbf{t} \not\preccurlyeq \mathbf{t}_1$. Let $\mathbf{t}' \in \mathbb{N}^k$ such that $\mathbf{t}' \preccurlyeq \mathbf{t}_2$ and hence $\mathbf{t}' \preccurlyeq \mathbf{t}_1$. By property (ii) of a well-suited partial order and since $\succcurlyeq$ is well-suited for $(\mathbb{Z}^k, \mathcal{E}^k_{\mathbb{Z}}, \succcurlyeq)$, there exists $\mathbf{t}'' \in \mathbb{Z}^k$ such that $\mathbf{t}'' \preccurlyeq \mathbf{t}_2$, $\mathbf{t}' \preccurlyeq \mathbf{t}''$ and $\mathbf{t} \preccurlyeq \mathbf{t}''$. By construction, we have $\mathbf{t}'' \in \mathbb{N}^k$ since $\mathbf{t}' \preccurlyeq \mathbf{t}''$. Moreover, we have $\mathbf{t}'' \preccurlyeq \mathbf{t}_1$ since $dcone(\mathbf{t}_2) \subseteq dcone(\mathbf{t}_1)$ in $(\mathbb{N}^k, E^k, \succcurlyeq)$ and hence $\mathbf{t} \preccurlyeq \mathbf{t}_1$ since $\mathbf{t} \preccurlyeq \mathbf{t}''$. That makes a contradiction with the fact that $\mathbf{t} \not\preccurlyeq \mathbf{t}_1$ and we conclude that if $expand(\mathbf{t}_2) \subseteq expand(\mathbf{t}_1)$, then $\mathbf{t}_1 \succcurlyeq \mathbf{t}_2$.

We prove the property (ii) as follows. Let $\mathbf{t} \in \mathcal{E}^k$ and $\mathbf{t}_1, \mathbf{t}_2$ two tuples such that $\mathbf{t}_1, \mathbf{t}_2 \in \mathbb{N}^k$, $\mathbf{t} \succcurlyeq \mathbf{t}_1$ and $\mathbf{t} \succcurlyeq \mathbf{t}_2$. We construct $\mathbf{t}_3$ as follows. For all $1 \leq i \leq k$, we have $t_{3i} = max(t_{1i}, t_{2i})$ where $max(c_1, c_2) = c_1$ if $c_1 \geq c_2$, otherwise $max(c_1, c_2) = c_2$. By construction, we have $\mathbf{t_3} \in \mathbb{N}^k$, $\mathbf{t} \succcurlyeq \mathbf{t}_3$, $\mathbf{t}_3 \succcurlyeq \mathbf{t}_1$ and $\mathbf{t}_3 \succcurlyeq \mathbf{t}_2$. □

As in the case of upward closed sets, if we restrict our attention to downward closed sets in $\mathbb{N}^k$, we obtain the following proposition. As a corollary, we have that any downward closed set in $\mathbb{N}^k$ has an unique finite minimal generator set.

Proposition 8 $(\mathbb{N}^k, \mathcal{E}^k, \succcurlyeq)$ is well-formed.

Proof. From Proposition 7, we have that $\succcurlyeq$ is well-suited for $(\mathbb{N}^k, \mathcal{E}^k, \succcurlyeq)$.

Let us now prove that any downward closed set of tuples in $\mathbb{N}^k$ has a finite generator set in $\mathbb{N}^k \cup \mathcal{E}^k$. Suppose that it is not the case and that the downward closed set $D \subseteq \mathbb{N}^k$ does not have a finite set of generators in $\mathbb{N}^k \cup \mathcal{E}^k$.

Let us construct a generator set of D. We first construct two sets.

1. The first set is the set $S_{lim} \subseteq \mathcal{E}^k$. First, let us define $S = \{\mathbf{t} \in \mathcal{E}^k \mid dcone(\mathbf{t}) \subseteq D\}$. We define S_{lim} such that $S_{lim} = \{\mathbf{t} \in S \mid \nexists \mathbf{t}' \in S$ with $\mathbf{t} \neq \mathbf{t}'$ and $\mathbf{t}' \succcurlyeq \mathbf{t}\}$. So S_{lim} is the set of maximal elements in S. Let us prove that for all $\mathbf{t} \in S$, there is $\mathbf{t}' \in S$ such that $\mathbf{t}' \succcurlyeq \mathbf{t}$

and $\mathbf{t}'' \not\succ \mathbf{t}'$ for all $\mathbf{t}'' \in S$, i.e. all the elements in S are covered by a maximal element in S. Hence for all $\mathbf{t} \in S$ there is $\mathbf{t}' \in S_{lim}$ with $\mathbf{t}' \succcurlyeq \mathbf{t}$.

Suppose that it is not the case. There exists $\mathbf{t} \in S$ such that for all $\mathbf{t}' \succ \mathbf{t}$ with $\mathbf{t}' \in S$ there exists $\mathbf{t}'' \in S$ with $\mathbf{t}'' \succ \mathbf{t}'$. Notice that by transitivity we have $\mathbf{t}'' \succ \mathbf{t}$ and hence (by applying the reasoning again) we can construct an infinite sequence $\mathbf{t}_1\mathbf{t}_2\ldots$ with $\mathbf{t}_i \in S$ and $\mathbf{t}_i \prec \mathbf{t}_{i+1}$ for all $i \geq 1$, and $\mathbf{t}_1 = \mathbf{t}'$.

Let $Inf(\mathbf{l})$ be the number of dimensions of $\mathbf{l}$ that are equal to $+\infty$. As Inf is bounded by k, we may define $I = max\{Inf(\mathbf{t}') \mid \mathbf{t}' \succcurlyeq \mathbf{t}, \mathbf{t}' \in S\}$ and choose $\mathbf{t}'$ such that $Inf(\mathbf{t}') = I$. Then, for all $i \geq 1$ we also have that $Inf(\mathbf{t}_i) = I$, and the infinite components are the same.

Let us construct $\mathbf{l}$ such that $l_i = +\infty$ if $t'_i = +\infty$ or there does not exist $c \in \mathbb{N}$ such that for all $j \geq 1$ we have $t_{ji} \leq c$. Otherwise, $l_i = c$ such that there is $j \in \mathbb{N}$ such that for all $j' \geq j$ we have $t_{j'i} = c$.

By construction, we have $\mathbf{l} \succcurlyeq \mathbf{t}$ and $Inf(\mathbf{l}) > I$ since for at least one dimension, the corresponding components are not finally constant. Hence $\mathbf{l} \notin S$.

However, we can prove that $dcone(\mathbf{l}) = \cup_{i\geq 1} dcone(\mathbf{t}_i)$, and $\cup_{i\geq 1} dcone(\mathbf{t}_i) \subseteq D$. As that property of $\mathbf{l}$ implies that $\mathbf{l} \in S$, this will make a contradiction.

Suppose that it is not the case and we have $dcone(\mathbf{l}) \neq \cup_{i\geq 1} dcone(\mathbf{t}_i)$. We consider two cases.

- There exists $\mathbf{l}' \in dcone(\mathbf{l})$ but $\mathbf{l}' \notin \cup_{i\geq 1} dcone(\mathbf{t}_i)$. In that case, there exists $1 \leq i \leq k$ such that for all $j \geq 1$ we have $l'_i > t_{ji}$ (remember that the sequence of the $\mathbf{t}_j$'s is strictly increasing). This means that $\mathbf{t}_1\mathbf{t}_2\ldots$ does not increase infinitely often on the dimension i, hence l_i is equal to the maximal value appearing in $\mathbf{t}_1\mathbf{t}_2\ldots$ on the dimension i. We conclude that there exists $\mathbf{t}_j$ $(j \geq 1)$ such that $t_{ji} = l_i$. As $l'_i \leq l_i$, we have that $l'_i \leq t_{ji}$ and that makes a contradiction with the fact that we have $l'_i > t_{ji}$ for all $j \geq 1$;
- there exists $\mathbf{l}' \in \cup_{i\geq 1} dcone(\mathbf{t}_i)$ but $\mathbf{l}' \notin dcone(\mathbf{l})$. In that case, there exists $1 \leq j \leq k$ with $l'_j > l_j$. This means that we have $l_j \neq +\infty$ and l_j is equal to the maximal value appearing in $\mathcal{S}$ on the dimension j. Hence, there is no $i \geq 1$ with $t_{ij} > l_j$ and we conclude that $l'_j \leq l_j$. That makes a contradiction with the fact that $l'_j > l_j$.

In both cases, we have a contradiction. Hence we conclude that $dcone(\mathbf{l}) = \cup_{i\geq 1} dcone(\mathbf{t}_i)$.

2. the second set is the set of maximal elements in D. We define $S_{max} = \{\mathbf{t} \in D \mid \nexists \mathbf{t}' \in D$ with $\mathbf{t}' \neq \mathbf{t}$ and $\mathbf{t}' \succcurlyeq \mathbf{t}\}$.

Let $S_{gen} = S_{lim} \cup S_{max}$. We first prove that the elements of S_{gen} are incomparable. By construction, we have that all the elements in S_{lim} are pairwise incomparable and all the elements in S_{max} are pairwise incomparable. Notice that each element in S_{lim} has at least one dimension equal to $+\infty$ and $S_{max} \subseteq \mathbb{N}^k$, hence there is no $\mathbf{t}_1 \in S_{max}$ and $\mathbf{t}_2 \in S_{lim}$ such that $\mathbf{t}_1 \succcurlyeq \mathbf{t}_2$.

Suppose that there is $\mathbf{t}_1 \in S_{max}$ and $\mathbf{t}_2 \in S_{lim}$ such that $\mathbf{t}_2 \succcurlyeq \mathbf{t}_1$ and let us construct a tuple $\mathbf{t} \in \mathbb{N}^k$ as follows. $\mathbf{t}$ is defined such that $t_i = t_{2i}$ if $t_{2i} \in \mathbb{N}$, otherwise $t_i = t_{1i} + 1$. By construction, we have that $\mathbf{t} \in D$ (as $\mathbf{t} \prec \mathbf{t}_2$), $\mathbf{t} \succ \mathbf{t}_1$ and hence $\mathbf{t}_1$ should be not in S_{max}. We conclude that all the elements of S_{max} are incomparable with the elements of S_{lim} and hence all the elements in S_{gen} are pairwise incomparable.

Finally, we prove that S_{gen} is a generator set of D as follows. Suppose that it is not the case. Notice that $dcones(S_{gen}) \subseteq D$ by construction. Indeed, all the elements of S_{max} are elements of D and each element in S_{lim} generates a subset of D.

Suppose that there exists $\mathbf{t} \in D$ but $\mathbf{t} \notin dcones(S_{gen})$. We consider two cases:

1. There exists an infinite sequence $\mathcal{S} = \mathbf{t}_1\mathbf{t}_2\ldots$ with $\mathbf{t}_1 = \mathbf{t}$ and for all $i \geq 1$ we have $\mathbf{t}_i \in D$ and $\mathbf{t}_i \prec \mathbf{t}_{i+1}$. Let us construct the tuple $\mathbf{t}'$ such that $t'_i = c$ if there is $l \geq 1$ with $t_{il'} = c$ for all $l' \geq l$, otherwise $t'_i = +\infty$. As D is downward closed and the elements of the sequence $\mathcal{S}$ are in D, we have that $dcones(\mathbf{t}') \subseteq D$. Moreover, there exists at least one dimension that increases infinitely often in $\mathcal{S}$ and hence there is at least one i such that $t'_i = +\infty$. So, $\mathbf{t}' \in S$ and there exists $\mathbf{t}'' \in S$ with $\mathbf{t}' \preceq \mathbf{t}''$ and $\mathbf{t}'' \in S_{lim}$. By construction we have $\mathbf{t} \prec \mathbf{t}'$ and we conclude that $\mathbf{t} \in dcones(S_{gen})$. This makes a contradiction with the fact that $\mathbf{t} \notin dcones(S_{gen})$.

2. There does not exist an infinite sequence $\mathcal{S} = \mathbf{t}_1\mathbf{t}_2\ldots$ with $\mathbf{t}_1 = \mathbf{t}$ and for all $i \geq 1$ we have $\mathbf{t}_i \in D$ and $\mathbf{t}_i \prec \mathbf{t}_{i+1}$. In that case, there exists $\mathbf{t}' \in D$ with $\mathbf{t} \preccurlyeq \mathbf{t}'$ and $\mathbf{t}'' \not\succ \mathbf{t}'$ for all $\mathbf{t}'' \in D$. Hence, $\mathbf{t}' \in S_{max}$, $\mathbf{t} \preccurlyeq \mathbf{t}'$ and we conclude that $\mathbf{t} \in dcones(S_{gen})$. This makes a contradiction with the fact that $\mathbf{t} \notin dcones(S_{gen})$.

In both cases, we obtain a contradiction and we conclude that there is no $\mathbf{t} \in D$ with $\mathbf{t} \notin dcones(S_{gen})$ and hence S_{gen} is a generator set of D.

By hypothesis, S_{gen} is infinite. So, we can construct an infinite sequence $\mathbf{t}_1\mathbf{t}_2\ldots$ with $\mathbf{t}_i \in S_{gen}$ and $\mathbf{t}_i \neq \mathbf{t}_j$ for all $i \neq j$. Following Lemma 9, $\preccurlyeq$ is a well quasi ordering on the elements in $(\mathbb{N} \cup \{+\infty\})^k$ and there is $i < j$ such that $\mathbf{t}_i \preccurlyeq \mathbf{t}_j$. This makes a contradiction with the fact that all the tuples in S_{gen} are pairwise incomparable.

Finally, we conclude that all the downward closed sets $D \subseteq \mathbb{N}^k$ have a finite generator set in $(\mathbb{N} \cup \{+\infty\})^k$. □

Corollary 6 *Any downward closed set in* $(\mathbb{N}^k, \mathcal{E}^k, \succcurlyeq)$ *has a unique finite minimal generator set.*

4.2 $\mathcal{U}$-logic and $\mathcal{D}$-logic

In this section we will introduce the logics $\mathcal{U}$ and $\mathcal{D}$ that we use to define upward-closed sets and downward-closed sets of tuples in $\mathbb{Z}^k$. These languages are defined on top of a subset of linear integer arithmetic constraints that have the form $x \geq c$ or $x \leq c$ where c is an integer constant. The complete languages are obtained by adding *disjunction* and *conjunction*. In the following, we will talk about formulas (without specifying the logic) when we refer without distinction to $\mathcal{U}$-formula or $\mathcal{D}$-formula.

Formally, let $V = \{x_1, \ldots, x_k\}$ be a finite set of variables. We first present the following grammar.

$$\Phi \quad ::= \quad x_i \bullet c \mid \Phi \wedge \Phi \mid \Phi \vee \Phi$$

where $c \in \mathbb{Z} \cup \{-\infty, +\infty\}$.

Replacing $\bullet$ by $\geq$ we obtain the grammar defining the set of $\mathcal{U}$-formulas. In the same way, replacing $\bullet$ by $\leq$ we obtain the grammar defining the set of $\mathcal{D}$-formulas.

Formulas are interpreted over $\mathbb{Z}^k$. In the following we will use the tuple $\mathbf{t} = \langle t_1, \ldots, t_k \rangle \in \mathbb{Z}^k$ to denote the valuation of the variables $\langle x_1, \ldots, x_k \rangle$ in which x_i is mapped to t_i for $1 \leq i \leq k$. Furthermore, we will use $\mathcal{U}_k$ ($\mathcal{D}_k$) to denote the set of $\mathcal{U}$-formulas ($\mathcal{D}$-formulas) defined over k variables. In the same way, $\mathcal{U}_k$-formula ($\mathcal{D}_k$-formula) refer to $\mathcal{U}$-formula ($\mathcal{D}$-formula) over k variables.

Definition 31 (Satisfaction Relation) Given a formula Φ, and a valuation $\mathbf{t}$, we define the *satisfaction relation* $\models$ as follows:

- let $\Phi = x_i \bullet c$ with $\bullet \in \{\geq, \leq\}$, then $\mathbf{t} \models \Phi$ if and only if $t_i \bullet c$;
- let $\Phi = \Phi_1 \wedge \Phi_2$, then $\mathbf{t} \models \Phi$ if and only if $\mathbf{t} \models \Phi_1$ and $\mathbf{t} \models \Phi_2$;
- let $\Phi = \Phi_1 \vee \Phi_2$, then $\mathbf{t} \models \Phi$ if and only if $\mathbf{t} \models \Phi_1$ or $\mathbf{t} \models \Phi_2$.

An *extended valuation* is a tuple $\mathbf{t} = \langle t_1, \ldots, t_k \rangle$ with $t_i \in \mathbb{Z} \cup \{-\infty, +\infty\}$. The satisfaction relation is naturally extended to extended valuations, with $+\infty + c = +\infty$ and $-\infty + c = -\infty$ and $+\infty \geq c \geq -\infty$ for any $c \in \mathbb{Z}$.

Definition 32 (Denotation of Formulas) Given a formula Φ, its *denotation* $[\![\Phi]\!]$ is defined as the set of all valuations $\mathbf{t}$ such that $\mathbf{t} \models \Phi$.

Definition 33 (Subsumption Relation) Given two $\mathcal{U}_k$-formulas ($\mathcal{D}_k$-formulas) Φ_1 and Φ_2, Φ_1 is *subsumed* by Φ_2, written $\Phi_1 \models \Phi_2$, whenever $[\![\Phi_1]\!] \subseteq [\![\Phi_2]\!]$.

As an example, let $k = 3$ and consider the formula $\Phi = ((x_1 \geq 3 \vee x_2 \geq 6) \wedge x_3 \geq 3)$. The denotation $[\![\Phi_1]\!]$ contains the tuples $\langle 3, 0, 4 \rangle$ and $\langle -4, 7, 4 \rangle$, whereas it does not contain the tuple $\langle 0, 1, 4 \rangle$. The formula $\Psi = x_1 \geq 1 \vee (x_2 \geq 3 \wedge x_3 \geq 2)$ subsumes Φ.

We say that two $\mathcal{U}_k$-formulas ($\mathcal{D}_k$-formulas) Φ_1 and Φ_2 are equivalent, written $\Phi_1 \equiv \Phi_2$, if their denotations coincide.

Definition 34 (Disjunctive form) A $\mathcal{U}$-formula in *disjunctive form* over the variables $\{x_1, \ldots, x_k\}$ has the following form:

$$\bigvee_{i:1,\ldots,n} (x_1 \geq c_{i1} \wedge \ldots \wedge x_k \geq c_{ik}).$$

In the same way, a $\mathcal{D}$-formula in *disjunctive form* over the variables $\{x_1, \ldots, x_k\}$ has the following form:

$$\bigvee_{i:1,\ldots,n} (x_1 \leq c_{i1} \wedge \ldots \wedge x_k \leq c_{ik}).$$

Any formula can always be reduced to an *equivalent disjunctive formula*. Note, in fact, that we can always add formulas like $x_i \geq -\infty$ in $\mathcal{U}$-formulas ($x_i \geq 0$ over non-negative integers) and $x_i \leq +\infty$ in $\mathcal{D}$-formula when x_i does not occur in a disjunct. A disjunctive form is obtained by distributing the conjuncts into the disjuncts as follows (and then adding basic constraints in the disjuncts):

$$\Phi_1 \wedge (\Phi_2 \vee \Phi_3) \equiv (\Phi_1 \wedge \Phi_2) \vee (\Phi_1 \wedge \Phi_3)$$

As an example, consider again the $\mathcal{U}$-formula Φ defined as

$$((x_1 \geq 3 \vee x_2 \geq 6) \wedge x_3 \geq 3)$$

It can be equivalently written as $\Phi_1 \vee \Phi_2$, where

$$\Phi_1 = x_1 \geq 3 \ \wedge \ x_2 \geq -\infty \wedge \ x_3 \geq 3$$

and

$$\Phi_2 = x_1 \geq -\infty \ \wedge \ x_2 \geq 6 \ \wedge \ x_3 \geq 3$$

Notice that there is several disjunctive forms of a formula. For instance, a second disjunctive form of Φ is $\Phi_1 \vee \Phi_2 \vee \Phi_3$, where Φ_1 and Φ_2 are defined as previously and

$$\Phi_3 = x_1 \geq 4 \ \wedge \ x_2 \geq 6 \ \wedge \ x_3 \geq 4$$

But notice that all the valuations satisfying Φ_3 also satisfy Φ_2.

As we will see later (see Corollary 8), the reduction of formula into disjunctive form cannot be done in polynomial time.

By construction, the denotation of a $\mathcal{U}$-formula ($\mathcal{D}$-formulas) φ is always an upward-closed (downward-closed) set of tuples, i.e., $cone([\![\varphi]\!]) = [\![\varphi]\!]$ ($dcone([\![\varphi]\!]) = [\![\varphi]\!]$).

If we restrict the denotations of $\mathcal{U}$-formula and $\mathcal{D}$-formulas to natural numbers, i.e. if we only consider valuations of variables over natural numbers, then the following proposition holds.

Proposition 9 (Completeness for $\mathbb{N}^k$) Given $I \subseteq \mathbb{N}^k$, if I is upward-closed (downward-closed), i.e., $cone(I) = I$ ($dcone(I) = I$), then there exists a $\mathcal{U}_k$-formula ($\mathcal{D}_k$-formula) φ such that $[\![\varphi]\!] = I$.

Proof. Let us prove the proposition for upward-closed sets. As I is an upward-closed subset of $\mathbb{N}^k$, it has a finite number of minimal elements $\{\mathbf{t}_1, \ldots, \mathbf{t}_n\}$ (see Corollary 5). Thus, we can associate with every tuple $\mathbf{t}_i = \langle t_{i1}, \ldots, t_{ik}\rangle$ the formula $\varphi_i \equiv x_1 \geq t_{i1} \wedge \ldots \wedge x_k \geq t_{ik}$. Clearly, we have that $cone(\mathbf{t}_i) = [\![\varphi_i]\!]$. Thus, the formula φ_I defined as $\varphi_I \equiv \varphi_1 \vee \ldots \vee \varphi_n$ is such that $I = [\![\varphi_I]\!]$.

The same reasoning can be applied for downward-closed set using Corollary 6. □

To ease the notation, in the sequel we will use Φ, Ψ, etc. to denote *arbitrary* formulas, and φ, ψ, etc. to denote formulas in disjunctive form. We now define the notion of *generator* of a disjunctive formula.

Definition 35 (Generators) The set of *generators* of a *disjunctive* $\mathcal{U}_k$-formula ($\mathcal{D}_k$-formula) φ, called $gen(\varphi)$, is the set of tuples $\langle c_1, \ldots, c_k\rangle$ with $c_i \neq +\infty$ ($c_i \neq -\infty$) for any $1 \leq i \leq k$ such that $x_1 \geq c_1 \wedge \ldots \wedge x_k \geq c_k$ ($x_1 \leq c_1 \wedge \ldots \wedge x_k \leq c_k$) is one of the disjuncts in φ.

We define the minimal elements of a $\mathcal{U}$-formula as follows.

Definition 36 (Minimal Elements) The set of *minimal elements* $min(\varphi)$ of a *disjunctive* $\mathcal{U}$-formula φ consists of the minimal elements (with respect to the component-wise ordering $\preccurlyeq$.) of $gen(\varphi)$.

In the same way, we define the maximal elements of a $\mathcal{D}$-formula as follows.

Definition 37 (Maximal Elements) The set of *maximal elements* $max(\varphi)$ of a *disjunctive* $\mathcal{D}$-formula φ consists of the maximal (with respect to the component-wise ordering $\preccurlyeq$) elements of $gen(\varphi)$.

Finally, we say that a disjunctive $\mathcal{U}$-formula ($\mathcal{D}$-formula) is in *normal form* whenever $gen(\varphi) = min(\varphi)$ ($gen(\varphi) = max(\varphi)$) . The generators in $gen(\varphi) \setminus min(\varphi)$ ($gen(\varphi) \setminus max(\varphi)$) are called *redundant*.

As an example, consider the formula $\varphi = (x \geq 1 \wedge y \geq 2) \vee (x \geq 3 \wedge y \geq 1) \vee (x \geq 2 \wedge y \geq 0)$. Then, $gen(\varphi) = \{\langle 1,2\rangle, \langle 3,1\rangle, \langle 2,0\rangle\}$, and $min(\varphi) = \{\langle 1,2\rangle, \langle 2,0\rangle\}$, i.e., φ is *not* in normal form. Note that, given a disjunctive $\mathcal{U}$-formula φ, $[\![\varphi]\!]$ coincides with the union of the denotations $cone(\mathbf{t})$ computed over all $\mathbf{t} \in min(\varphi)$, i.e.,

$$[\![\varphi]\!] = \bigcup_{\mathbf{t} \in min(\varphi)} cone(\mathbf{t}). \tag{4.1}$$

For a general $\mathcal{U}$-formula Ψ we define $min(\Psi)$ as the minimal set of tuples with components in $\mathbb{Z} \cup \{-\infty\}$ satisfying

$$[\![\Psi]\!] = \bigcup_{\mathbf{t} \in min(\Psi)} cone(\mathbf{t}). \tag{4.2}$$

Similarly, for a disjunctive $\mathcal{D}$-formula φ, we have

$$[\![\varphi]\!] = \bigcup_{\mathbf{t} \in max(\varphi)} dcone(\mathbf{t}). \tag{4.3}$$

and for a general $\mathcal{D}$-formula Ψ $max(\Psi)$ is defined as the minimal set of tuple with components in $\mathbb{Z} \cup \{+\infty\}$ satisfying

$$[\![\Psi]\!] = \bigcup_{\mathbf{t} \in max(\Psi)} dcone(\mathbf{t}). \tag{4.4}$$

Notice that for a general $\mathcal{U}$-formula Φ or a general $\mathcal{D}$-formula Ψ, the sets $min(\Phi)$ and $max(\Psi)$ are always finite and unique. Indeed, let us consider the $\mathcal{U}$-formula Φ (the case of $\mathcal{D}$-formulas can be directly deduce from the case of $\mathcal{U}$-formulas) and let $\Phi_1 \vee \ldots \vee \Phi_n$ where $\Phi_i = x_1 \geq c_{i1} \vee \ldots x_k \geq c_{ik}$ $(1 \leq i \leq n)$ be a corresponding disjunctive form of Φ. $min(\Phi)$ is constructed by first computing the set $\{\langle c_{ij}, \ldots, c_{ik}\rangle \mid 1 \leq i \leq n, 1 \leq j \leq k\}$ and then removes from this set the elements that are not minimal with respect to $\preccurlyeq$.

Following Proposition 4, the sets constructed in this way are minimal generator sets of the upward closed sets (downward closed sets) described by the formulas. From Proposition 3 and Proposition 5 (Proposition 3 and Proposition 7 in the case of $\mathcal{D}$-formulas), we have that those minimal sets are unique.

Finally, we define the size of a formula Φ (noted $|\Phi|$) inductively as follows.
$|x_i \bullet c_i| = 1$ where $\bullet \in \{\leq, \geq\}$;
$|\phi_1 \bullet \phi_2| = 1 + |\phi_1| + |\phi_2|$ where $\bullet \in \{\vee, \wedge\}$;

4.3 Operations on $\mathcal{U}$-formulas and $\mathcal{D}$-formulas in Disjunctive Form

We present in this section some results and algorithms on the formulas of the logics $\mathcal{U}$ and $\mathcal{D}$. More precisely, results and algorithms on $\mathcal{U}$-formulas and $\mathcal{D}$-formulas are similar. For this reason, we only present those related to $\mathcal{U}$-formulas in this section. All the results and all the algorithms can be adapted to $\mathcal{D}$-formulas by replacing $x \geq c$ by $x \leq c$, $max(t_i, t'_i)$ by $min(t_i, t'_i)$, $min(\Phi)$ by $max(\Phi)$ and $\mathbf{t} \preccurlyeq \mathbf{t}'$ by $\mathbf{t}' \preccurlyeq \mathbf{t}$.

4.3.1 Disjunction

Clearly, formulas in disjunctive form are closed under disjunction. Furthermore, given two formulas in normal form φ_1 and φ_2 we can build a formula ψ in normal form such that $[\![\psi]\!] = [\![\varphi_1]\!] \cup [\![\varphi_2]\!]$ in time $\mathcal{O}(|\varphi_1| \times |\varphi_2|)$ (the time needed to remove redundancies (see section 4.3.6)).

4.3.2 Conjunction

Given the $\mathcal{U}_k$-formulas in disjunctive form φ and ψ, the disjunctive formula for $\varphi \wedge \psi$ is defined as follows:

$$\bigvee_{\mathbf{t} \in gen(\varphi), \mathbf{t}' \in gen(\psi)} \bigwedge_{i=1}^{k} x_i \geq max(t_i, t'_i),$$

i.e., $gen(\varphi \wedge \psi) = \{\mathbf{s} \mid \exists \mathbf{t} \in gen(\varphi), \mathbf{t}' \in gen(\psi) \text{ and } s_i = max(t_i, t'_i),\ 1 \leq i \leq k\}$. Intuitively, a valuation $\mathbf{t}$ of the conjunction of two formulas must satisfy a disjunct of the first formula and a disjunct of the second one. This means that for the first disjunct we must have $t_i \geq c_{1i}$ if we have the constraint $x_i \geq c_{1i}$ for all $1 \leq i \leq k$ in the disjunct and for the second disjunct we have $t_i \geq c_{2i}$ if we have the constraint $x_i \geq c_{2i}$ for all $1 \leq i \leq k$ in the disjunct, i.e. $t_i \geq max(c_{1i}, c_{2i})$ for all $1 \leq i \leq k$.

The formula $\varphi \wedge \psi$ can be built in time $\mathcal{O}(|\varphi_1| \times |\varphi_2|)$. In general the result of this operation might not be in normal form, even if φ and ψ are.

4.3.3 Satisfiability

Given a (extended) valuation $\mathbf{t}$, we have that $\mathbf{t} \models \varphi$ holds if and only if there exists $\mathbf{t}' \in gen(\varphi)$ such that $\mathbf{t}' \preccurlyeq \mathbf{t}$. Thus, checking $\mathbf{t} \models \varphi$ can be done in time $\mathcal{O}(|\varphi|)$. Checking if a valuation $\mathbf{t}$ satisfies an arbitrary $\mathcal{U}$-formula Φ can also be done in time $\mathcal{O}(|\Phi|)$. In fact, we simply have to apply the definition of $\models$ by inspecting the structure of Φ.

4.3.4 Subsumption

Let φ and ψ be two $\mathcal{U}_k$-formulas in disjunctive form. We can check $\varphi \models \psi$ in time $\mathcal{O}(|\varphi| \times |\psi|)$. In fact, $\varphi \models \psi$ holds if and only if for all $\mathbf{t} \in gen(\varphi)$ there exists $\mathbf{t}' \in gen(\psi)$ such that $\mathbf{t}' \preccurlyeq \mathbf{t}$. As stated in the following theorem, the subsumption test is much harder for generic $\mathcal{U}$-formulas.

Theorem 7 Given two $\mathcal{U}_k$-formulas Φ and Ψ, checking that $\Phi \models \Psi$ is coNP-hard in the size of the formulas, even when they are evaluated on $\mathbb{N}^k$.

Proof. We use a reduction from the *validity* problem (see e.g. [Joh90]) for propositional formulas in disjunctive normal form (DNF):

> *Given a propositional formula $F = D_1 \vee \ldots \vee D_n$ where each disjunct D_i is a conjunction of literals over the set of propositional variables $p_1, \ldots, p_m$, is F true for all valuations?*

The size of a propositional formula F in DNF (noted $|F|$) is inductively defined as follows.
$|p| = 1$ where p is a propositional variable,
$|\neg p| = 1$ where p is a propositional variable,
$|F_1 \vee F_2| = 1 + |F_1| + |F_2|$,
$|F_1 \wedge F_2| = 1 + |F_1| + |F_2|$.

The validity problem for propositional formulas in DNF is coNP-complete [Joh90]. The reduction works as follows. Given a propositional formula F, we build a $\mathcal{U}$-formula $\Phi_{val(F)}$ (polynomial in the size of F) that represents all the valuations that make F true. Then, we reduce validity for F to the subsumption problem $\Phi_{val(true)} \models \Phi_{val(F)}$. Without loss of generality, we assume that *all variables* occur in each disjunct of F either as p, $\neg p$ or $any(p)$. The latter means that p can take any value (it is equivalent to say that p *does not* occur in the disjunct). The formula $true$ is equivalent to $any(p_1) \wedge \ldots \wedge any(p_m)$. To encode the valuation for p_i, we use two variables from V, namely v_i and w_i. We use the formula $\Phi_{val(p_i)} = (v_i \geq 1 \wedge w_i \geq 0)$ to check if p_i evaluates to $true$; $\Phi_{val(\neg p_i)} = (v_i \geq 0 \wedge w_i \geq 1)$ to check if p_i evaluates to $false$. Note that neither $\Phi_{val(p_i)} \models \Phi_{val(\neg p_i)}$ nor $\Phi_{val(\neg p_i)} \models \Phi_{val(p_i)}$. We use the disjunction $\Phi_{val(any(p_i))} = (v_i \geq 1 \wedge w_i \geq 0) \vee (v_i \geq 0 \wedge w_i \geq 1)$ when p_i can be evaluated either to $true$ or $false$. The previous encoding allows us to keep the size of $\Phi_{val(F)}$ polynomial in the size of F.

Formally, the formula $\Phi_{val(F)}$ is defined by induction on F as follows:

- $\Phi_{val(D_1 \vee D_2)} = \Phi_{val(D_1)} \vee \Phi_{val(D_2)}$;
- $\Phi_{val(C_1 \wedge C_2)} = \Phi_{val(C_1)} \wedge \Phi_{val(C_2)}$;
- $\Phi_{val(p_i)} = (v_i \geq 1 \wedge w_i \geq 0)$;
- $\Phi_{val(\neg p_i)} = (v_i \geq 0 \wedge w_i \geq 1)$;
- $\Phi_{val(any(p_i))} = (v_1 \geq 0 \wedge w_1 \geq 1) \vee (v_1 \geq 1 \wedge w_1 \geq 0)$.

$\Phi_{val(true)}$ is constructed in the same way.

Based on this construction, we obtain that

F is valid if and only if $\Phi_{val(true)} \models \Phi_{val(F)}$.

Since the reduction is *polynomial*, we obtain that the subsumption problem for $\mathcal{U}$-formulas is coNP-hard. □

To prove coNP easiness, we first need the following lemma.

Lemma 10 Let Φ be an $\mathcal{U}_k$-formula and let $const(\Phi)$ be the set of integral constants in Φ. Then, for all $\mathbf{t} \in min(\Phi)$, we have $t_i \in const(\Phi) \cup \{-\infty\}$ for any $1 \leq i \leq k$.

Proof. We prove the proposition by contraposition. Let $\mathbf{t} \in (\mathbb{Z} \cup \{-\infty\})^k$ be a valuation with $t_i \notin const(\Phi) \cup \{-\infty\}$ for a fixed i. We will show that $\mathbf{t} \notin min(\Phi)$. We first isolate constraints on x_i imposed by the components t_j of $\mathbf{t}$ for $j \neq i$. This is done by constructing the formula Φ' as follows. Replace every constraint of the form $x_j \geq c$ by $true$ if $t_j \geq c$, by $false$ otherwise for any $j \neq i$. After simplification, Φ' can be of three forms.

1. $\Phi' \equiv false$. We should have at least one $c \in const(\Phi)$ with $c \neq -\infty$. Otherwise, all the constraints $x_j \geq c_j$ become $true$ and we should have $\Phi' \equiv true$. Consider the tuple $\mathbf{t}'$ constructed from $\mathbf{t}$ as follows. Let $c \in \mathbb{Z}$ be the smallest constant in Φ different from

$-\infty$. We have $t'_h = t_h$ if $t_h \in \mathbb{Z}$, otherwise $t'_h = c - 1$ for all h such that $1 \leq h \leq k$ (in particular $t'_i = t_i$). By construction, we have $\mathbf{t}' \in \mathbb{Z}^k$ and $\mathbf{t} \preccurlyeq \mathbf{t}'$. Moreover, we have $\mathbf{t}' \not\models \Phi$ because for each h $(1 \leq h \leq k)$ such that $t_h = -\infty$ we have that a constraint $x_h \geq c_h$ in Φ is not satisfied by $\mathbf{t}$ if and only if $\mathbf{t}'$ does not satisfy $x_h \geq c_h$. We conclude that $\mathbf{t} \notin min(\Phi)$;

2. $\Phi' \equiv true$. In that case, consider $\mathbf{t}'$ constructed from $\mathbf{t}$ by replacing t_i by $-\infty$, $\mathbf{t}'$ is such that $\mathbf{t}' \models \Phi$ and as $\mathbf{t}' \prec \mathbf{t}$ we conclude that $\mathbf{t} \notin min(\Phi)$;

3. Φ' is a boolean combination of constraints of the form $x_i \geq c$. In that case, we can further simplify Φ' to obtain an equivalent formula Φ'' as follows. In a bottom-up manner, we replace $x_i \geq c_1 \wedge x_i \geq c_2$ by $x_i \geq max(c_1, c_2)$ and $x_i \geq c_1 \vee x_i \geq c_2$ by $x_i \geq min(c_1, c_2)$. So, we finally obtain $\Phi'' \equiv x_i \geq d$, and we know that $d \in const(\Phi)$ by construction. Then there are two cases. Either $d > t_i$ and $\mathbf{t} \not\models \Phi$ and we conclude that $\mathbf{t} \notin min(\Phi)$ or $d < t_i$ and we can construct $\mathbf{t}'$ from $\mathbf{t}$ by replacing t_i by d. We have $\mathbf{t}' \models \Phi$ and as $\mathbf{t}' \prec \mathbf{t}$ we conclude that $\mathbf{t} \notin min(\Phi)$. □

Now, we are ready to prove the following result.

Theorem 8 Given two $\mathcal{U}_k$-formulas Φ and Ψ, checking that $\Phi \models \Psi$ is coNP-easy in the size of the formulas.

Proof. We prove that the complement of the subsumption problem is in NP.
As $[\![\Phi]\!]$ and $[\![\Psi]\!]$ are upward-closed, we have that

$$\Phi \not\models \Psi \text{ iff } \exists \mathbf{t} \in min(\Phi) : \mathbf{t} \not\models \Psi$$

Following Lemma 10, we can decide that $\Phi \not\models \Psi$ by guessing a tuple $\mathbf{t}$ in the set of tuples $S_\Phi = \{\mathbf{t} | \mathbf{t} = \langle t_1, \ldots, t_k \rangle, t_i \in const(\Phi) \cup \{-\infty\}, 1 \leq i \leq k\}$ whose size is bounded by $(|\Phi|+1)^k$ and check in deterministic polynomial time that $\mathbf{t} \models \Phi$ and $\mathbf{t} \not\models \Psi$. □
As a consequence of the previous results, we have the following corollary.

Corollary 7 Given two $\mathcal{U}_k$-formulas Φ and Ψ, checking that $\Phi \models \Psi$ is coNP-complete in the size of the formulas even if the formulas are evaluated on $\mathbb{N}^k$.

As the subsumption problem is polynomial when formulas are in disjunctive form, following Corollary 7, we conclude that the construction of the disjunctive form cannot be done in polynomial time.

Corollary 8 Given a $\mathcal{U}_k$-formulas Φ, there is no polynomial algorithm that computes an equivalent $\mathcal{U}_k$-formula in disjunctive form.

Proof. Suppose that it is not the case and there exists such a polynomial algorithm. A possible algorithm to decide if $\Phi \models \Psi$ consists to reduce Φ and Ψ into equivalent disjunctive $\mathcal{U}_k$-formulas Φ' and Ψ' and then check in polynomial time if $\Phi' \models \Psi'$. Hence, we have a polynomial time algorithm to decide if $\Phi \models \Psi$. That makes a contradiction with the fact that such an algorithm does not exist. We conclude that there is no algorithm to compute the disjunctive form of a $\mathcal{U}_k$-formula formula. □

4.3.5 Minimization

Given two $\mathcal{U}_k$-formulas in disjunctive form $\varphi = \varphi_1 \vee ... \vee \varphi_n$ and $\psi = \psi_1 \vee ... \vee \psi_m$, the reduction in normal form of φ according to ψ, written $\varphi \div \psi$, is the formula obtained by removing from φ every disjunct φ_i for which there exists a disjunct ψ_j in ψ such that $\varphi_i \models \psi_j$.

For instance, $((x_1 \geq 2 \wedge x_2 \geq 2 \wedge x_3 \geq -\infty) \vee (x_1 \geq -\infty \wedge x_2 \geq -\infty \wedge x_3 \geq 2) \div (x_1 \geq -\infty \wedge x_2 \geq 1 \wedge x_3 \geq -\infty)$ is the formula $x_1 \geq -\infty \wedge x_2 \geq -\infty \wedge x_3 \geq 2$ because $x_1 \geq 2 \wedge x_2 \geq 2 \wedge x_3 \geq -\infty \models x_1 \geq -\infty \wedge x_2 \geq 1 \wedge x_3 \geq -\infty$ but we do not have $x_1 \geq -\infty \wedge x_2 \geq -\infty \wedge x_3 \geq 2 \models x_1 \geq -\infty \wedge x_2 \geq 1 \wedge x_3 \geq -\infty$.

Intuitively, if φ and ψ represent parts of a collection of state representations, e.g. the set of new states (i.e. the frontier) and the already computed states during symbolic state exploration, see Section 5.2.1, then the operation $\div$ allows us to remove all redundant generators from φ with respect to the generators in ψ. This corresponds to minimize the frontier (the set of new states that must be treated) with respect to the set of already computed (and treated) states.

The formula $\varphi \div \psi$ can be constructed in time $\mathcal{O}(|\varphi| \times |\psi|)$. The denotation of $\varphi \div \psi$ is defined as

$$[\![\varphi \div \psi]\!] = \{\mathbf{c} \in cone(\mathbf{t}) \mid \mathbf{t} \in [\![\varphi]\!],\ \mathbf{t} \notin [\![\psi]\!]\}$$

The $\div$ operation can be seen as a kind of difference : only subsets of valuations generated by a disjunct of φ that is not subsumed by ψ are kept.

4.3.6 Reduction in normal form

Given a disjunctive formula $\varphi = \varphi_1 \vee \ldots \vee \varphi_n$ we can reduce it in normal form by eliminating all formulas φ_i such that the corresponding generator is not a minimal element of the set defined by φ. The reduction can be done in time $\mathcal{O}(|\varphi|^2)$.
Notice that the normal form of φ is different from $\varphi \div \varphi$. Indeed, as all the disjuncts φ_i appears on the two sides of the $\div$ operator, the resulting formula of $\varphi \div \varphi$ is $false$.

In the following sections we will introduce special data structures and we will show how to use them to handle large sets of *generators* of formulas in disjunctive form.

4.4 Symbolic Representation of $\mathcal{U}$-formulas and $\mathcal{D}$-formulas

In practice, formulas in disjunctive form are often used. This is mainly due to the fact that they are simpler to manipulate (see previous section)

However, formulas in disjunctive form can become be very large when they are used to describe sets of states of systems with counters over non-negative integer. This is what we call the *symbolic state explosion* problem. To avoid this problem, we need efficient data structures to represent upward and downward closed sets of tuples compactly. Since in the following we are interested in the representation of sets of tuples in $\mathbb{N}^k$, we restrict in this section the denotations of $\mathcal{U}$-formulas and $\mathcal{D}$-formulas to tuples over natural numbers. In other words, we consider all along this section that $\mathcal{U}$-formulas and $\mathcal{D}$-formulas represent upward and downward closed sets of tuples in $\mathbb{N}^k$.

First, we present two data structures that have been studied in the literature and that can be used to represent upward and downward closed sets of tuples in $\mathbb{N}^k$. The first one, called *Interval Decision Diagram*, is based on the BDD data structure [Bry86]. The second one is the *Natural Decision Diagrams* [WB95] that allow us to encode any set definable with Presburger formulas. We show for both data structures that some upward and downward closed sets are encoded very inefficiently, i.e. some sets cannot be encoded with IDDs or NDDs with a size polynomial in the size of the corresponding $\mathcal{U}$-formula ($\mathcal{D}$-formula). Then, we present a new data structure, called *Covering Sharing Tree*, based on the *Sharing Tree* data structure [ZC95]. Finally, we study the complexity of the operations on covering sharing trees needed to symbolically manipulate $\mathcal{U}$-formulas and $\mathcal{D}$-formulas.

4.4.1 Deterministic Automata

Interval Decision Diagrams

The data structure called *Interval Decision Diagram* (IDD for short) has been proposed by K. Strehl and L. Thiele in [ST98] to encode efficiently infinite sets of tuples of positive integer including upward and downward closed sets of tuples.

Let an interval $[a, b]$ over $\mathbb{N}$ (where $a, b \in \mathbb{N} \cup \{+\infty\}$ with $a \leq b$) be the set $\{n \in \mathbb{N} \mid a \leq n \leq b\}$ of all the natural numbers that are greater or equal to a and smaller or equal to b. Let a *partition* be a finite set of intervals over natural numbers $\{I_1, \ldots, I_l\}$ such that for any $1 \leq i, j \leq l$ $(i \neq j)$ we have $I_i \cap I_j = \emptyset$ and $\mathbb{N} = \cup_{1 \leq i \leq l} I_i$. We note the set of all the intervals over natural numbers as $S^{\mathcal{I}}$.

We specialise the definition of IDDs given in [ST98] to encode upward and downward closed sets of tuples in $\mathbb{N}^k$ as follows.

Definition 38 (k-Interval Decision Diagram (k-IDD)) A *k-Interval Decision Diagram* $\mathbf{I}$ is a directed acyclic graph with a set of non terminal nodes N and a set of terminal nodes $T = \{$ *accepted*, *refused* $\}$.

The set of non-terminal nodes N contains one special root node (without predecessors). There is a function $dim : N \to [1, k]$ that assigns a dimension to each non terminal node. We define the successor function $succ : N \to 2^{(N \cup T) \times S^{\mathcal{I}}}$, such that for any $n \in N$, we have $succ(n) = \{(n_1, I_1), \ldots, (n_l, I_l)\}$ with $l \geq 1$ where $\{I_1, \ldots I_l\}$ is a partition, $\{n_1, \ldots, n_l\}$ is the set of successors of n and either $n_i \in T$ or $dim(n) < dim(n_i)$ for any $1 \leq i \leq l$.

Graphically, we will represent non-terminal nodes of IDDs by circles containing their dimension. The terminal nodes are represented by rectangles containing their name (*accepted* or $refused$). The *succ* function is represented by arrows between nodes labelled by an interval. For instance, an arrow from the node n_1 to the node n_2 labelled by the interval $[a, b]$ means that $(n_2, [a, b]) \in succ(n_1)$.

Definition 39 (Path/co-Path) A *path* of the k-IDD $\mathbf{I}$ is a finite sequence of nodes and intervals $(n_1, I_1), \ldots, (n_l, I_l)$ where $n_1 = root$, for all $1 \leq i < l$ we have $(n_{i+1}, I_i) \in succ(n_i)$ and $(accepted, I_l) \in succ(n_l)$. A *co-path* of the k-IDD $\mathbf{I}$ is a finite sequence of nodes and intervals $(n_1, I_1), \ldots, (n_l, I_l)$ where $n_1 = root$, for all $1 \leq i < l$ we have $(n_{i+1}, I_i) \in succ(n_i)$ and $(refused, I_l) \in succ(n_l)$.

In the following, we say that a path or a co-path $(n_1, I_1), \ldots, (n_l, I_l)$ encodes a tuple $\mathbf{t}$ if $t_{dim(n_i)} \in I_i$ for all $1 \leq i \leq l$. The paths define a set S of tuples and the co-paths define its complement $\overline{S}$ as specified in the following definition.

Definition 40 (Denotation/co-Denotation) The *denotation* of a k-IDD $\mathbf{I}$, written *denote*($\mathbf{I}$) , is the set of tuples in $\mathbb{N}^k$ encoded by the paths of $\mathbf{I}$. Formally, $denote(\mathbf{I}) = \{\mathbf{t} \in \mathbb{N}^k |$

there is a path $(n_1, I_1), \ldots, (n_l, I_l)$ of $\mathbf{I}$ that encodes $\mathbf{t}\}$. In the same way, $co-denote(\mathbf{I}) = \{\mathbf{t} \in \mathbb{N}^k |$ there is a co-path $(n_1, I_1), \ldots, (n_l, I_l)$ of $\mathbf{I}$ that encodes $\mathbf{t}\}$.

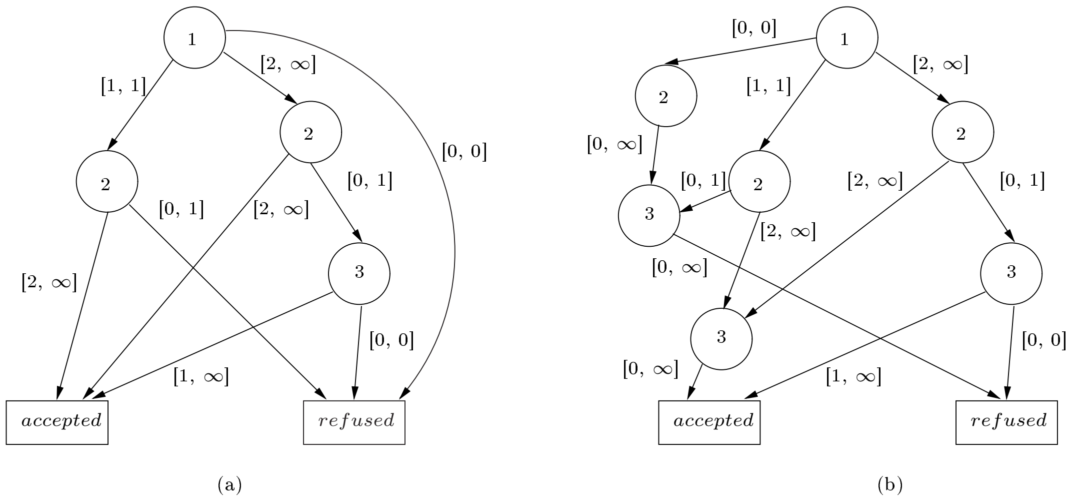

Figure 4.3: (a) An example of IDD, (b) the corresponding completed IDD.

IDDs allows us to represent sets of tuples over $\mathbb{N}^k$ described by a finite disjunction of conjunctions of basic formulas of the form $x \in [a, b]$ where $[a, b]$ is an interval, i.e. constraints of the form $\bigvee_{i=1\ldots n} \wedge_{j=1\ldots k} x_j \in [a_{ij}, b_{ij}]$. When the basic formulas are of the form $x \in [a, +\infty]$, we obtain constraints similar to $\mathcal{U}$-formulas, i.e. the constraints describe upward closed sets of tuples. Constraints are similar to $\mathcal{D}$-formulas, i.e. the constraints describe downward closed sets of tuples, when the basic constraints have the restricted form $x \in [0, b]$. All the upward and downward closed sets in $\mathbb{N}^k$ can be encoded using constraints of this kind and hence can be represented by IDDs. Indeed, suppose U to be an upward closed set. For each $\mathbf{t} \in min(U)$, we construct the conjuncts $C_{\mathbf{t}} = x_1 \in [t_1, +\infty] \wedge \ldots x_k \in [t_k, +\infty]$. Then, the constraints describing U is $\bigvee_{\mathbf{t} \in min(U)} C_{\mathbf{t}}$. A similar reasoning can be used to encode downward closed sets of tuples over natural numbers.

Intuitively, we encode $\mathcal{U}$-formulas and $\mathcal{D}$-formulas on variables $\{x_1, \ldots, x_k\}$ with k-IDDs by defining a bijection $f : \{1, \ldots, k\} \rightarrow \{1, \ldots, k\}$. This bijection allows us to associate the variable x_i $(1 \leq i \leq k)$ to each node n such that $f(dim(n)) = i$. Such a function f is called an *encoding strategy*. We use $denote(\mathbf{I})_f$ to represent the set of tuples encoded by $\mathbf{I}$ using the encoding strategy f, i.e. $denote(\mathbf{I})_f = \{\mathbf{t} \mid \exists \mathbf{t}' \in denote(\mathbf{I})$ such that $t_{f(i)} = t'_i$ for all $1 \leq i \leq k\}$.

For example, the 3-IDD of figure 4.3(a) encodes the set $\{\mathbf{t} \in \mathbb{N}^3 | (t_1 \geq 2 \wedge t_3 \geq 1) \vee (t_1 \geq 1 \wedge t_2 \geq 2)\}$. Associating the *ith* dimension to x_i, this 3-IDD encodes the $\mathcal{U}$-formula $x_1 \geq 2 \wedge x_3 \geq 1 \vee x_1 \geq 1 \wedge x_2 \geq 2$ on the set of variables $\{x_1, x_2, x_3\}$ that have the denotation $\{\mathbf{t} \in \mathbb{N}^3 \mid t_1 \geq 2 \wedge t_3 \geq 1 \vee t_1 \geq 1 \wedge t_2 \geq 2\}$.

Experiences show that k-IDDs allow to efficiently manipulate sets of tuples (see for instance [ST00]). However, sometimes, k-IDDs can be particularly inefficient when they are used to encode upward and downward closed sets as stated by Theorem 9. That theorem states that there exist $\mathcal{U}_k$-formulas ($\mathcal{D}_k$-formulas) φ in disjunctive form and encoding strategies f such that there is no IDDs encoding φ with f compactly.

Theorem 9 There is no polynomial $p(l)$ in l such that the following holds for all $\mathcal{U}_k$-formula ($\mathcal{D}_k$-formula) φ in disjunctive form.

For all the encoding strategies f over k variables, there exists an IDDs $\mathbf{I}_\varphi$ such that $denote(\mathbf{I}_\varphi)_f = [\![\varphi]\!]$ and the number of nodes of $\mathbf{I}_\varphi$ is bounded by $p(|\varphi|)$.

Proof. Let f be an encoding strategy. In the following, we note f_i instead of x_j when $f(i) = j$.

Let the formula in disjunctive form

$$\varphi = \bigvee_{i=1\ldots l} f_1 \geq 0 \wedge \ldots \wedge f_i \geq 1 \wedge f_{i+1} \geq 0 \wedge \ldots \wedge f_{l+i} \geq 1 \wedge f_{l+i+1} \geq 0 \wedge \ldots \wedge f_{2l} \geq 0.$$

This formula has a size polynomial in l because the number of disjuncts is equal to l. So, we prove that there does not exist a polynomial $p(l)$ such that for all $l \geq 1$ there exists an IDD $\mathbf{S}_\varphi$ with a number of nodes bounded by $p(l)$ and $denote(\mathbf{I}_\varphi)_f = [\![\varphi]\!]$.

This formula φ describes an upward closed set of tuples in $\mathbb{N}^{2l}$ that satisfy the following property. The set of variables is divided into two subsets: the set of the l first variables encoded into IDDs and the set of the l last variables encoded into IDDs. A valuation satisfies φ if and only if there is a variable in the first set encoded at the ith position in IDDs such that its value is greater or equal to 1 in the valuation and such that the variable of the second set encoded at the $(i+l)th$ position in IDDs is also greater or equal to 1 in the valuation.

Suppose that there exists a polynomial $p(k)$ such that there exists an IDD $\mathbf{I}_\varphi$ encoding φ with f ($denote(\mathbf{I}_\varphi)_f = [\![\varphi]\!]$) with a number of nodes bounded by $p(l)$. Without loss of generality, we suppose in the following that for each non terminal node n of $\mathbf{I}_\varphi$ with $dim(n) = i$ we have $succ(n) = \{(n_1, I_1), \ldots, (n_k, I_k)\}$ such that for all $1 \leq j \leq k$ we have $dim(n_j) = i+1$ if $i < 2l$. If the considered IDD does not satisfy this property, we add intermediate nodes (see Figure 4.3(b)). The number of nodes to add in the IDD to satisfy this assumption is polynomial in l. Indeed, for each node of the IDD we have to add at most $2l-1$ nodes (one for each variable) and hence we can construct a polynomial $p'(k)$ from $p(k)$ that bounds the number of nodes of the IDDs after adding the intermediate nodes.

Let us define the sets $C(\mathcal{V})$ with $\mathcal{V} \in 2^{\{1,\ldots,l\}}$ such that

$$C(\mathcal{V}) = \{\mathbf{t} \in \mathbb{N}^{2l} | \forall 1 \leq i \leq l : t_i > 0 \text{ if and only if } i \in \mathcal{V}\}$$

For each $\mathcal{V} \in 2^{\{1,\ldots,l\}}$ ($\mathcal{V} \neq \emptyset$) we have $C(\mathcal{V}) \cap [\![\mathbf{I}_\varphi]\!] \neq \emptyset$ and for all $\mathcal{V}_1, \mathcal{V}_2 \in 2^{\{1,\ldots,l\}}$ ($\mathcal{V}_1 \neq \mathcal{V}_2$) we have $C(\mathcal{V}_1) \cap C(\mathcal{V}_2) = \emptyset$.

As there are $2^l - 1$ sets $C(\mathcal{V})$ with $\mathcal{V} \neq \emptyset$ and the number of nodes of $\mathbf{I}_\varphi$ is bounded by $p(l)$, there exist two tuples $\mathbf{m}, \mathbf{n} \in [\![\mathbf{I}_\varphi]\!]$ such that $\mathbf{m} \in C(\mathcal{V}_1)$ and $\mathbf{n} \in C(\mathcal{V}_2)$ with $\mathcal{V}_1 \neq \mathcal{V}_2$, $\mathbf{m}$ is encoded by the path $(n_1, I_1) \ldots (n_l, I_l)(n_{l+1}, I_{l+1}) \ldots (n_{2l}, I_{2l})$, and $\mathbf{n}$ is encoded by the path $(n'_1, I'_1) \ldots (n'_l, I'_l)(n'_{l+1}, I'_{l+1}) \ldots (n'_{2l}, I'_{2l})$ with $n_{l+1} = n'_{l+1}$, for l great enough.

Without loss of generality, suppose that $m_k > 0$ and $n_k = 0$ for some $1 \leq k \leq l$ and let us define $\mathbf{m}' \in \mathbb{N}^{2l}$ as a tuple similar to $\mathbf{m}$ on the lth first dimensions and with the lth last dimensions equal to 0 except for the $(l+k)th$ dimension, i.e. for all $1 \leq i \leq l$ we have $m'_i = m_i$ and for all $l+1 \leq i \leq 2l$ we have $m'_i > 0$ if and only if $i = l+k$. By construction, we have $\mathbf{m}' \in [\![\mathbf{I}_\varphi]\!]$. As $\mathbf{I}_\varphi$ is deterministic, $\mathbf{m}'$ is encoded by a path of the form $(n_1, I_1) \ldots (n_l, I_l)(n_{l+1}, I''_{l+1})(n''_{l+2}, I''_{l+2}) \ldots (n''_{2l}, I''_{2l})$. We conclude that $\mathbf{I}_\varphi$ has also the path $(n'_1, I'_1) \ldots (n'_l, I'_l)(n'_{l+1}, I''_{l+1})(n''_{l+2}, I''_{l+2}) \ldots (n''_{2l}, I''_{2l})$ that encodes the tuple $\mathbf{n}'$ such that for all $1 \leq i \leq l$ we have $n'_i = n_i$ and for all $l+1 \leq i \leq 2l$ we have $n'_i = m'_i$. This means that for all $l+1 \leq i \leq 2l$ we have $n'_i > 0$ if and only if $i = l+k$ and $n_k = 0$. Let $\mathbf{n}''$ be such that $n''_{f(i)} = n'_i$ for all $1 \leq i \leq 2l$. We conclude that $\mathbf{n}'' \notin [\![\varphi]\!]$ because for all i such that $l+1 \leq i \leq 2l$ with $i \neq l+k$ we have $n''_{f(i)} = 0$ and $n''_{f(k)} = 0$. We obtain a contradiction with the fact that $denote(\mathbf{I}_\varphi)_f = [\![\varphi]\!]$.

We conclude that the number of nodes of $\mathbf{I}_\varphi$ is not bounded by $p(l)$, for l great enough.

A similar proof can be exhibited for $\mathcal{D}$-formula starting from the formula

$$\varphi' = \bigvee_{i=1\ldots l} f_1 \leq +\infty \wedge \ldots \wedge f_i \leq 0 \wedge f_{i+1} \leq +\infty \wedge \ldots \wedge f_{l+i} \leq 0 \wedge f_{l+i+1} \leq +\infty \wedge \ldots \wedge f_{2l} \leq +\infty.$$

This formula describes the set of tuples in $\mathbb{N}^{2l}$ where there is at least two dimensions i and $i+l$ $(1 \leq i \leq l)$ equal to 0.

Suppose that there exists a polynomial $p(k)$ such that there exists an IDD $\mathbf{I}_{\varphi'}$ encoding φ' with f $(denote(\mathbf{I}_{\varphi'})_f = [\![\varphi']\!])$ with a number of nodes bounded by $p(l)$.

The sets $C(\mathcal{V})$ with $\mathcal{V} \in 2^{\{1,\ldots,l\}}$ are defined as

$$C(\mathcal{V}) = \{\mathbf{t} \in \mathbb{N}^{2l} | \forall 1 \leq i \leq l : t_i = 0 \text{ if and only if } i \in \mathcal{V}\}$$

For each $\mathcal{V} \in 2^{\{1,\ldots,l\}}$ $(\mathcal{V} \neq \emptyset)$ we have $C(\mathcal{V}) \cap [\![\mathbf{I}_{\varphi'}]\!] \neq \emptyset$ and for all $\mathcal{V}_1, \mathcal{V}_2 \in 2^{\{1,\ldots,l\}}$ $(\mathcal{V}_1 \neq \mathcal{V}_2)$ we have $C(\mathcal{V}_1) \cap C(\mathcal{V}_2) = \emptyset$.

As there are $2^l - 1$ sets $C(\mathcal{V})$ and the number of nodes of $\mathbf{I}_{\varphi'}$ is bounded by $p(l)$, there exist two tuples $\mathbf{m}, \mathbf{n} \in [\![\mathbf{I}_{\varphi'}]\!]$ such that $\mathbf{m} \in C(\mathcal{V}_1)$ and $\mathbf{n} \in C(\mathcal{V}_2)$ with $\mathcal{V}_1 \neq \mathcal{V}_2$, $\mathbf{m}$ is encoded by the path $(n_1, I_1) \ldots (n_l, I_l)(n_{l+1}, I_{l+1}) \ldots (n_{2l}, I_{2l})$, and $\mathbf{n}$ is encoded by the path $(n'_1, I'_1) \ldots (n'_l, I'_l)(n'_{l+1}, I'_{l+1}) \ldots (n'_{2l}, I'_{2l})$ with $n_{l+1} = n'_{l+1}$, for l great enough.

Without loss of generality, suppose that $m_k = 0$ and $n_k > 0$ for some $1 \leq k \leq l$ and let us define $\mathbf{m}' \in \mathbb{N}^{2l}$ as a tuple similar to $\mathbf{m}$ on the lth first dimensions and with the lth last dimensions equal to 1 except for the $(l+k)th$ dimension, i.e. for all $1 \leq i \leq l$ we have $m'_i = m_i$ and for all $l+1 \leq i \leq 2l$ we have $m'_i = 0$ if and only if $i = l+k$. By construction, we have that $\mathbf{m}' \in [\![\mathbf{I}_{\varphi'}]\!]$. As $\mathbf{I}_{\varphi'}$ is deterministic, $\mathbf{m}'$ is encoded by a path of the form $(n_1, I_1) \ldots (n_l, I_l)(n_{l+1}, I''_{l+1})(n''_{l+2}, I''_{l+2}) \ldots (n''_{2l}, I''_{2l})$. We conclude that $\mathbf{I}_{\varphi'}$ has also the path $(n'_1, I'_1) \ldots (n'_l, I'_l)(n'_{l+1}, I''_{l+1})(n''_{l+2}, I''_{l+2}) \ldots (n''_{2l}, I''_{2l})$ that encodes the tuple $\mathbf{n}'$ such that for all $1 \leq i \leq l$ we have $n'_i = n_i$ and for all $l+1 \leq i \leq 2l$ we have $n'_i = m'_i$. This means that for all $l+1 \leq i \leq 2l$ we have $n'_i = 0$ if and only if $i = l+k$ and $n_k > 0$. Let $\mathbf{n}''$ be such that $n''_{f(i)} = n'_i$ for all $1 \leq i \leq 2l$. We conclude that $\mathbf{n}'' \notin [\![\varphi']\!]$ because for all i such that $l+1 \leq i \leq 2l$ with $i \neq l+k$ we have $n''_{f(i)} > 0$ and $n''_{f(k)} > 0$. We obtain a contradiction with the fact that $denote(\mathbf{I}_{\varphi'})_f = [\![\varphi']\!]$.

We conclude that the number of nodes of $\mathbf{I}_{\varphi'}$ is not bounded by $p(l)$, for l great enough.

□

The previous theorem is a direct adaptation of Theorem 28 of [Gan02].

Natural Decision Diagrams

Natural decision diagrams are finite deterministic automata that accept binary encodings of tuples of natural numbers. Such automata are presented for instance in [BC96, WB95] and have been implemented into several libraries and tools, like in the LASH toolkit [LAS] and the FAST model checking tool [BFLP03].

Let w be a sequence $\mathbf{b}_1 \ldots \mathbf{b}_n$ where $\mathbf{b}_i$ $(1 \leq i \leq n)$ is a tuple of m dimensions of the form $\langle b_{i1}, \ldots, b_{im} \rangle$ where $b_{ij} \in \{0,1\}$. Let $\mathcal{T} : (\{0,1\}^m)^* \to \mathbb{N}^m$ be the function that associates to

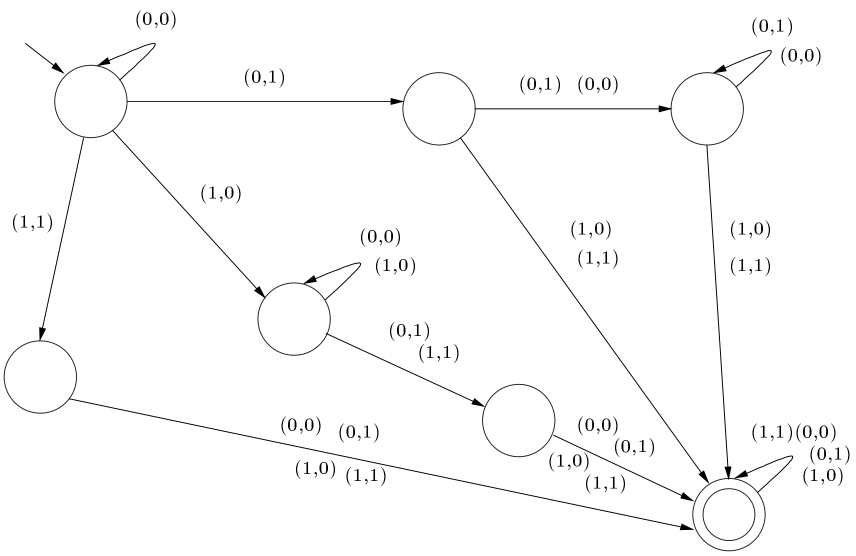

Figure 4.4: The NDD that represents the set of pairs $\langle x_1, x_2 \rangle$ such that $x_1 \geq 1 \wedge x_2 \geq 2$.

each sequence of vectors of *bits* w a tuple $\mathbf{t} \in \mathbb{N}^m$ such that for all $1 \leq i \leq m$ we have

$$t_i = \sum_{j=1\ldots n} 2^{n-j} b_{ji}.$$

For instance, $\mathcal{T}$ will return $\langle 7, 1 \rangle$ when applied on the sequence of pairs of bits $\langle 1,0 \rangle \langle 1,0 \rangle \langle 1,1 \rangle$. Notice that $\mathcal{T}(\mathbf{b}_1 \ldots \mathbf{b}_n) = \mathcal{T}(\mathbf{0} \ldots \mathbf{0} \mathbf{b}_1 \ldots \mathbf{b}_n)$ where $\mathbf{0}$ is the tuple where all the bits are equal to 0.

Definition 41 A *Natural Decision Diagram* (NDD for short) that encodes the set $S \subseteq \mathbb{N}^k$ is the minimal deterministic automaton $A = \langle Q, \Sigma, \sigma, s_0, F \rangle$ where Q is a set of states, $\Sigma = \{0,1\}^m$ is an alphabet, $\sigma = Q \times \Sigma \rightarrow Q$ is a transition relation, $s_0 \in Q$ is the initial state and $F \subseteq Q$ is a set of terminal states. Moreover, the language $L(A)$ accepted by A is the set of all the words $w \in (\{0,1\}^m)^*$ such that $\mathcal{T}(w) \in S$.

Figure 4.4 shows the NDD that represents the set of pairs $\langle x_1, x_2 \rangle$ such that $x_1 \geq 1 \wedge x_2 \geq 2$. The states are represented by circles, and transitions are represented by labelled arrows between states. To clarify the picture, if we have k transitions from the state n_1 to n_2 with the labels $l_1, \ldots, l_k$ in the NDD, they are graphically represented by one arrow labelled by the labels $l_1, \ldots, l_k$. The initial state is depicted with a small arrow and the accepting state is represented with a second circle inside.

Notice that some subsets of $\mathbb{N}^m$ cannot be encoded using NDDs. However, all the sets of tuples over natural numbers that are described by at least one formula of the Presburger logic can be represented by a NDD. Moreover, all the upward and downward closed sets of tuples in $\mathbb{N}^k$ can be described by Presburger formulas. Hence, all the upward and downward closed set in $\mathbb{N}^k$ can be represented by NDDs.

As in the case of IDDs, we use encoding functions $f : \{1, \ldots, k\} \rightarrow \{1, \ldots, k\}$ to encode $\mathcal{U}_k$-formula ($\mathcal{D}_k$-formula) into NDDs. Given an encoding function f and a NDD A, we define $L_f(A)$ as the set of tuples encoded by the NDD using the encoding strategy f, i.e. $L_f(A) = \{\mathbf{t} \mid \exists \mathbf{t}' \in L(A) \text{ such that } t_{f(i)} = t'_i \text{ for all } 1 \leq i \leq k\}$.

In [WB00], P. Wolper and B. Boigelot consider automata over the alphabet $\{0,1\}$ to encode sets of tuples in $\mathbb{N}^m$. These automata are similar to NDDs except that they read the *ith* bit of each dimensions in a defined order before reading the $i+1$ bits instead of reading the *ith* of each dimension in one step. For instance, a transition from a state n_1 to n_2 labeled by $(1,0)$ in a NDD will be replaced by one intermediate state n and two transitions. The first transition goes from n_1 to n and is labelled by 1, and the second one goes from n to n_2 and is labelled by 0. From n_1, the first bit read will correspond to a bit of the first variable and the second one will correspond to a bit of the second variable. Notice that when the automata reaches n_2 from n_1, it has read exactly the same bit for each variable than the NDD.

So those automata are very similar to NDDs and Theorem 10 can be easily adapted to the automata of [WB00]. That theorem states that there exist $\mathcal{U}$-formulas ($\mathcal{D}$-formulas) φ in normal form such that there is no NDD that encodes efficiently φ. Contrary to Theorem 9, this result holds for any encoding strategy f.

Theorem 10 There is no polynomial $p(l)$ in l such that the following holds for all $\mathcal{U}_k$-formula ($\mathcal{D}_k$-formula) in disjunctive form φ.

There exists an encoding strategy f and a NDD A_φ such that $L_f(A_\varphi) = [\![\varphi]\!]$ and the number of nodes of A_φ is bounded by $p(|\varphi|)$.

Proof. Let the formula

$$\varphi = \bigvee_{i=1\ldots l} x_1 \geq 0 \wedge \ldots \wedge x_i \geq 1 \wedge x_{i+1} \geq 0 \wedge \ldots \wedge x_{l+i} \geq 1 \wedge x_{l+i+1} \geq 0 \wedge \ldots \wedge x_{2l} \geq 0.$$

φ is the same formula than the formula used in the proof of Theorem 9. So, as the formula has a size polynomial in l, we prove that there does not exist a polynomial $p(l)$ such that for all $l \geq 1$ there exists an encoding strategy f and a NDD A_φ with a number of nodes bounded by $p(l)$ such that $L(A_\varphi)_f = [\![\varphi]\!]$.

Assume that there exists a polynomial $p(k)$ such that for all $l \geq 1$ there exists a NDD that encodes φ with a number of nodes bounded by $p(l)$.

Let A_φ be a NDD encoding the set of tuples satisfying φ for $l \geq 1$. Suppose that this NDD has a number of states bounded by $p(l)$. Let f be the encoding strategy used to encode φ into A_φ.

Let us define the sets $C(\mathcal{V})$ with $\mathcal{V} \in 2^{\{1,\ldots,l\}}$ such that

$$C(\mathcal{V}) = \{w \in (\{0,1\}^{2l})^* \mid w = \mathbf{b}_1 \ldots \mathbf{b}_n \text{ such that } \forall 1 \leq i \leq 2l : \sum_{j=1..n} b_{ji} > 0 \text{ iff } f(i) \in \mathcal{V}\}$$

By construction, we have that for all $\mathcal{V}_1, \mathcal{V}_2 \in 2^{\{1,\ldots,l\}}$ with $\mathcal{V}_1 \neq \mathcal{V}_2$ we have $C(\mathcal{V}_1) \cap C(\mathcal{V}_2) = \emptyset$.

As A_φ has a number of states bounded by $p(l)$ and there are 2^l sets $C(\mathcal{V})$, there exist two words $w_1 \in C(\mathcal{V}_1), w_2 \in C(\mathcal{V}_2)$ with $\mathcal{V}_1 \neq \mathcal{V}_2, \mathcal{V}_1 \neq \emptyset$ and $\mathcal{V}_2 \neq \emptyset$ such that the paths in A_φ recognizing w_1 and w_2 from $\mathbf{s}_0$ lead to the same state $\mathbf{s}$, for l great enough.

Without loss of generality, we have that $i \in \mathcal{V}_1$ and $i \notin \mathcal{V}_2$ for some $1 \leq i \leq l$. As $w_1 \in C(\mathcal{V}_1)$, w_1 is of the form $\mathbf{b}_1 \ldots \mathbf{b}_{n_1}$ where at least one $\mathbf{b}_j$ has its dimension corresponding to x_i equal to 1. We conclude that reading $\mathbf{b}$ from $\mathbf{s}$ such that $b_j = 1$ if $f(j) = i + l$, $b_j = 0$ otherwise, after reading w_1 corresponds to a word encoding a tuple where the dimensions corresponding to x_i and x_{i+l} are greater or equal to 1. Let $\mathbf{t}$ be such that $\mathbf{t}' = \mathcal{T}(w_1 \cdot \mathbf{b})$ and $t_{f(i)} = t'_i$ for all $1 \leq i \leq 2l$. Hence, $\mathbf{t} \in [\![\varphi]\!]$ and A_φ reaches a final state (allowing it to accept the word).

As $w_2 \in C(\mathcal{V}_2)$, w_2 is of the form $\mathbf{b}'_1 \ldots \mathbf{b}'_{n_2}$ where all the $\mathbf{b}'_j$'s have their dimension corresponding to x_i equal to 0. Moreover, as the path corresponding to w_2 in A_φ leads to the state $\mathbf{s}$, we conclude that the word $w_3 = \mathbf{b}'_1 \ldots \mathbf{b}'_{n_2}\mathbf{b}$ is accepted by the NDD. This word is such that $\mathbf{t} = \mathcal{T}(w_3)$ where for all j such that $l+1 \leq f(j) \leq 2l$ we have $t_j = 0$ if $f(j) \neq i+l$ and $t_j = 1$ if $f(j) = i+l$. Moreover, if $f(j) = i$, then $t_j = 0$. As this tuple does not satisfies φ, we obtain a contradiction.

We conclude that A_φ has a number of nodes that is not bounded by $p(l)$.

To prove the result for $\mathcal{D}$-formulas, we proceed in the same way by starting from the formula

$$\varphi' = \bigvee_{i=1\ldots l} x_1 \leq +\infty \wedge \ldots \wedge x_i \leq 0 \wedge x_{i+1} \leq +\infty \wedge \ldots \wedge x_{l+i} \leq 0 \wedge x_{l+i+1} \leq +\infty \wedge \ldots \wedge x_{2l} \leq +\infty.$$

Assume that there exists a polynomial $p(k)$ such that for all $l \geq 1$ there exists a NDD that encodes φ' with a number of nodes bounded by $p(l)$.

Let $A_{\varphi'}$ be a NDD encoding the set of tuples satisfying φ' for $l \geq 1$. Suppose that this NDD has a number of states bounded by $p(l)$. Let f be the encoding strategy used to encode φ' into $A_{\varphi'}$.

In that case, the sets $C(\mathcal{V})$ are defined as

$$C(\mathcal{V}) = \{w \in (\{0,1\}^{2l})^* \mid w = \mathbf{b}_1 \ldots \mathbf{b}_n \text{ such that } \forall 1 \leq i \leq 2l : \sum_{j=1..n} b_{ji} = 0 \text{ iff } f(i) \notin \mathcal{V}\}$$

By construction, for all $\mathcal{V}_1, \mathcal{V}_2 \in 2^{\{1,\ldots,l\}}$ with $\mathcal{V}_1 \neq \mathcal{V}_2$ we have $C(\mathcal{V}_1) \cap C(\mathcal{V}_2) = \emptyset$.

As $A_{\varphi'}$ has a number of states bounded by $p(l)$ and there are 2^l sets $C(\mathcal{V})$, there exist two words $w_1 \in C(\mathcal{V}_1), w_2 \in C(\mathcal{V}_2)$ with $\mathcal{V}_1 \neq \mathcal{V}_2, \mathcal{V}_1 \neq \emptyset$ and $\mathcal{V}_2 \neq \emptyset$ such that the paths in $A_{\varphi'}$ recognizing w_1 and w_2 from $\mathbf{s}_0$ lead to the same state $\mathbf{s}$, for l great enough.

Without loss of generality, we have that $i \notin \mathcal{V}_1$ and $i \in \mathcal{V}_2$ for some $1 \leq i \leq l$. As $w_1 \in C(\mathcal{V}_1)$, w_1 is of the form $\mathbf{b}_1 \ldots \mathbf{b}_{n_1}$ where there is no $\mathbf{b}_j$ $(1 \leq j \leq n_1)$ that has its dimension corresponding to x_i equal to 1. We conclude that reading $\mathbf{b}$ from $\mathbf{s}$ such that $b_j = 1$ if $f(j) \neq i+l$, $b_j = 0$ otherwise, after reading w_1 corresponds to a word encoding a tuple where the dimensions corresponding to x_i and x_{i+l} are equal to 0. Let $\mathbf{t}$ be such that $\mathbf{t}' = \mathcal{T}(w_1 \cdot \mathbf{b})$ and $t_{f(i)} = t'_i$ for all $1 \leq i \leq 2l$. Hence, $\mathbf{t} \in [\![\varphi']\!]$ and $A_{\varphi'}$ reaches a final state (allowing it to accept the word).

As $w_2 \in C(\mathcal{V}_2)$, w_2 is of the form $\mathbf{b}'_1 \ldots \mathbf{b}'_{n_2}$ where there is at least one j with $1 \leq j \leq n_2$ and $\mathbf{b}'_j$ has its dimension corresponding to x_i equal to 1. Moreover, as the path corresponding to w_2 in $A_{\varphi'}$ leads to the state $\mathbf{s}$, we conclude that the word $w_3 = \mathbf{b}'_1 \ldots \mathbf{b}'_{n_2}\mathbf{b}$ is accepted by $A_{\varphi'}$. This word is such that $\mathbf{t} = \mathcal{T}(w_3)$ where for all j such that $l+1 \leq f(j) \leq 2l$ we have $t_j = 1$ if $f(j) \neq i+l$ and $t_j = 0$ if $f(j) = i+l$. Moreover, if $f(j) = i$, then $t_j > 0$. As this tuple does not satisfies φ', we obtain a contradiction.

We conclude that $A_{\varphi'}$ has a number of nodes that is not bounded by $p(l)$. $\square$

The previous theorem also holds if the natural decision diagrams read the *least significant bits first*; i.e. if $\mathcal{T}$ is defined such that $\mathcal{T}(w) = \mathbf{t}$ with

$$t_i = \sum_{j=1\ldots n} 2^{j-1} b_{ji}.$$

In the next section, we present new data structures called *Upward Closed Covering Sharing Tree* and *Downward Closed Covering Sharing Tree*. As we will see in the next section, those data structures, contrary to IDD and NDD, never have a size exponential in the size of the corresponding $\mathcal{U}$-formula or $\mathcal{D}$-formula. Moreover, we prove that there exist some upward and downward closed sets that are efficiently encoded using those data structures.

4.4.2 Non-Deterministic Automata

We present in this section data structures to encode upward closed and downward closed sets of tuples and called covering sharing trees. Those data structures are based on the *sharing trees* data structure presented by D. Zampuniéris and B. Le Charlier in [ZC95].

This section is organized as follows. We first present sharing trees that are deterministic data structures allowing us to encode a finite set of tuples. Then, we show how to extend them to encode upward and downward closed sets.

Sharing Trees

In this section, we specialize the original definition of [ZC95] as follows.

Definition 42 (k-Sharing Tree) A *k-sharing tree* is a rooted directed acyclic graph $(N, val, V, succ)$, where

- $N = \{root\} \cup N_1 \ldots \cup N_k \cup \{end\}$ is the finite set of *nodes*; N_i is the set of nodes of *layer* i and, by convention, $N_0 = \{root\}$ and $N_{k+1} = \{end\}$. Moreover, all the N_i's are pairwise disjoint;
- V is a (possibly infinite) set of values. No assumption is given to the domain V except that it must contain two special values $\perp$ and $\top$ and the equality test must be defined on elements of V;
- $val : N \rightarrow V$ is a labeling function that assigns to each node a value of the domain V;
- $succ : N \rightarrow 2^N$ defines the successors of a node;
- Furthermore,
    1. $val(n) = \top$ if and only if $n = root$;
    2. $val(n) = \perp$ if and only if $n = end$;
    3. $succ(end)=\emptyset$;
    4. for $i : 0, \ldots, k$, for all $n \in N_i$, $succ(n) \subseteq N_{i+1}$ and $succ(n) \neq \emptyset$;
    5. for all $n \in N$, for all $n_1, n_2 \in succ(n)$, if $n_1 \neq n_2$ then $val(n_1) \neq val(n_2)$;
    6. for $i : 0, \ldots, k$, for all $n_1, n_2 \in N_i$ s.t. $n_1 \neq n_2$, if $val(n_1) = val(n_2)$ then $succ(n_1) \neq succ(n_2)$.

In other words, a k-sharing tree is an acyclic graph with a root and a terminal node such that:

- all the nodes of layer i have their successors in the layer $i + 1$ (cond. 4);
- a node cannot have two successors with the same label (cond. 5);

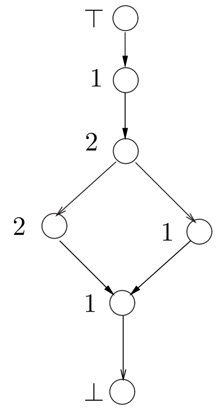

Figure 4.5: An example of k-sharing tree.

- two nodes with the same label in the same layer do not have the same set of successors (cond. 6).

Condition 5 and 6 ensure the *maximal sharing* of *prefixes* and *suffixes* among the paths (elements) of a k-sharing tree. This means that all the tuples that shares a same prefix of size i will be encoded by paths that share their i first nodes and there is no nodes that are the roots of isomorphic sub-trees.

By convention, the *empty k-sharing tree* is the sharing tree with only root and end nodes such that $succ(root) = \emptyset$ (notice that the particular empty k-sharing tree does not satisfy the condition 4 of Definition 42).

Definition 43 (Pre-sharing tree) $\mathbf{S}$ is a *pre-sharing tree* whenever it satisfies at least the conditions(1)-(4).

In the rest of the thesis we use $root^{\mathbf{S}}$, $end^{\mathbf{S}}$, $N^{\mathbf{S}}$, and $succ^{\mathbf{S}}$ to refer to the *root*, *set of nodes*, *successor relation* of the sharing-tree $\mathbf{S}$, respectively.

Definition 44 (Path) A *path* of a k-sharing tree is a sequence of nodes $\langle root, n_1, \ldots, n_k, end\rangle$ such that $n_{i+1} \in succ(n_i)$ for $i : 1, \ldots, k-1$, $n_1 \in succ(root)$ and, finally, $end \in succ(n_k)$.

Paths represent tuples of size k of values in V as specified in the following definition.

Definition 45 (Flat Denotation) The *flat denotation* of a k-sharing tree $\mathbf{S}$, denoted by $elem(\mathbf{S})$, is the set of elements represented by the paths in $\mathbf{S}$. Formally, the set $elem(\mathbf{S})$ consists of all tuples $\langle val(n_1), \ldots, val(n_k)\rangle$ such that $\langle root, n_1, \ldots, n_k, end\rangle$ is a path in $\mathbf{S}$.

By defining the domain V as $V = \mathbb{Z} \cup \{\top, \bot\}$, k-sharing trees can be used to encode finite sets of elements in $\mathbb{Z}^k$. For instance, Figure 4.5 shows an example of k-sharing tree that represents the set $\{\langle 1, 2, 2, 1\rangle, \langle 1, 2, 1, 1\rangle\}$ of tuples over integral numbers. Circles labelled by a value represent states with their associated value and arrows represent the successor relation, i.e. we have an arrow from n_1 to n_2 if $n_2 \in succ(n_1)$.

Definition 46 (Size of a k-sharing tree) The *size* of a k-sharing tree $\mathbf{S}$, denoted $|\mathbf{S}|$, is defined as the number of nodes plus the number of edges.

As shown in [Zam97], thanks to sharing, the cardinality of $elem(\mathbf{S})$ can be exponentially larger than the size of $\mathbf{S}$.

Given a node n of the i-th layer of a k-sharing tree $\mathbf{S}$, the sub-(sharing)tree $\mathbf{S}_n$ rooted at n is the $k-i+1$-sharing tree obtained as follows. We first isolate the graph rooted at n that consists of all the nodes reachable from n (this subgraph has $k-i+1$ layers and a terminal node). Then, we add a layer with the single root node and we set $succ(root) = \{n\}$. From the previous definition, $elem(\mathbf{S}_n)$ consists of all tuples $\langle m, m_{i+1}, \ldots, m_k \rangle$ obtained from the sub-path $\langle n, n_{i+1}, \ldots, n_k, end \rangle$ of $\mathbf{S}$ such that $m = val(n)$ and $m_j = val(n_j)$ when $j = i+1, \ldots, k$.

As shown in [ZC95], given a set F of tuples of size k, there exists a unique (modulo isomorphisms of graphs) k-sharing tree such that $elem(\mathbf{S}) = F$. In the same paper, the authors define algorithms for the basic operations on the flat denotations of k-sharing trees. Table 4.1 summarizes the complexity (in the size of k-sharing trees) of the set operations implemented via k-sharing trees. In the rest of the thesis we will often refer to operations like union, intersection, emptiness, containment, and equality test. In [ZC95], the authors showed that the operations in Table 4.1 can also be applied to pre-sharing trees that satisfy at least condition 5.

Operation	**Effect**	**Complexity**
emptiness test	$\mathsf{is\_empty}(\mathbf{S})$ *if and only if* $elem(\mathbf{S}) = \emptyset$	constant
membership test	$\mathsf{member}(\mathbf{t}, \mathbf{S})$ *if and only if* $\mathbf{t} \in elem(\mathbf{S})$	poly
containment test	$\mathsf{contained}(\mathbf{S}, \mathbf{T})$ *if and only if* $elem(\mathbf{S}) \subseteq elem(\mathbf{T})$	poly
union	$elem(\mathsf{union}(\mathbf{S}, \mathbf{T})) = elem(\mathbf{S}) \cup elem(\mathbf{T})$	poly
intersection	$elem(\mathsf{intersection}(\mathbf{S}, \mathbf{T})) = elem(\mathbf{S}) \cap elem(\mathbf{T})$	poly
minus	$elem(\mathsf{minus}(\mathbf{S}, \mathbf{T})) = elem(\mathbf{S}) \setminus elem(\mathbf{T})$	poly

Table 4.1: Complexity of the operations on k-sharing trees.

Covering Sharing Trees

As mentioned in the previous section, k-sharing trees can be used to compactly represent *finite sets* of tuples. Our goal, however, is to symbolically represent *upward-closed sets* and *downward-closed sets* of integer tuples. *Symbolically* here means that we want to represent upward and downward closed set of integer tuples *without enumerating* their elements (or the elements of their finite generator sets). For this purpose, we need to *lift* the denotation of a k-sharing tree to a *richer one*, in which the tuples stored in its paths represent *generators* of formulas in disjunctive form. In the next section, we will study the impact of the lifting from flat to rich denotations on the complexity of the symbolic operations. We first show, however, how to use k-sharing trees to represent $\mathcal{U}$-formulas and $\mathcal{D}$-formulas in disjunctive form.

Definition 47 (k-Covering Sharing Tree (k-CST)) A *k-Covering Sharing Tree* (k-CST for short) $\mathbf{S}$ is a k-sharing tree where a partial order $\sqsubseteq$ is defined on the elements of V. Moreover, V is of the form $V = D \cup E \cup \{\top, \bot\}$ where D is a domain of values and E is a set of limit elements.

In the following, we will always associate to D^k the set of limit elements $\mathcal{E}^k = \{\mathbf{t} \in (D \cup E)^k \mid$ there is at least one $1 \leq i \leq k$ with $t_i \in E\}$. We also consider $\preccurlyeq$ as the componentwise extension of $\sqsubseteq$ to tuples in $(D \cup E)^k$.

The set represented by covering sharing trees are closed sets. More precisely, we define the rich denotation of k-CSTs as follows.

Definition 48 (Rich Denotation) The *rich denotation* of the k-CST $\mathbf{S}$ is defined as follows

$$cones(\mathbf{S}) = \{\mathbf{t}' \in D^k \mid \exists \mathbf{t} \in elem(\mathbf{S}) : \mathbf{t} \preccurlyeq \mathbf{t}'\}.$$

The notions of pre-sharing tree and sub-sharing tree, defined at the end of the section presenting the k-sharing trees, naturally extend to k-CSTs (see pages 103-104).

As in the case of k-sharing trees (see page 113), by convention, we represent the empty set by a k-ucCST that only contains the root and end nodes with $succ(root) = \emptyset$.

Definition 49 (Irredundant k-CST) The k-CST $\mathbf{S}$ is *irredundant* if and only if there exists no $\mathbf{t} \in elem(\mathbf{S})$ such that $\mathbf{t}' \preccurlyeq \mathbf{t}$ for some $\mathbf{t}' \in elem(\mathbf{S})$ distinct from $\mathbf{t}$.

Following Proposition 4, when $\preccurlyeq$ is well-suited, the paths of irredundant k-CSTs correspond to the minimal finite generator sets of the closed sets they represent.

To encode $\mathcal{U}$-formulas and $\mathcal{D}$-formulas into k-CSTs, we define $D = \mathbb{N}$ in both cases, and respectively the relations $\leq$ and $\geq$ on natural numbers as total order. Moreover, $E = \emptyset$ in the case of $\mathcal{U}$-formulas and $E = \{+\infty\}$ in the case of $\mathcal{D}$-formulas. As usual, we use encoding functions $f : \{1, \ldots, k\} \to \{1, \ldots, k\}$ to encode $\mathcal{U}$-formulas and $\mathcal{D}$-formulas into k-CSTs. More precisely, encoding functions are bijections that associate a variable to each layer of k-CSTs. Given a k-CST, we define the set $elem(\mathbf{S})_f$ as the set $\{\mathbf{t} \mid \exists \mathbf{t}' \in elem(\mathbf{S})$ such that $t_{f(i)} = t'_i$ for all $1 \leq i \leq k\}$. $cones(\mathbf{S})_f$ is defined is the same way.

More formally, we define the *upward closed covering sharing trees* and the *downward closed covering sharing trees* as follows.

Definition 50 (Upward Closed Covering Sharing Tree (k-ucCST)) Let φ be a $\mathcal{U}_k$-formula in disjunctive form. The *Upward Closed Covering Sharing Tree* $\mathbf{S}_\varphi$ that represents φ with the encoding strategy f is a covering sharing tree where $D = \mathbb{N}$, $E = \emptyset$ and the considered order on $\mathbb{N}$ is $\leq$ such that $elem(\mathbf{S}_\varphi)_f = gen(\varphi)$.

Definition 51 (Downward Closed Covering Sharing Tree (k-dcCST)) Let φ be a $\mathcal{D}_k$-formula in disjunctive form. The *Downward Closed Covering Sharing Tree* $\mathbf{S}_\varphi$ that represents φ with the encoding strategy f is a covering sharing tree where $D = \mathbb{N}$, $E = \{+\infty\}$ and the considered order on $\mathbb{N} \cup \{+\infty\}$ is $\geq$ such that $elem(\mathbf{S}_\varphi)_f = gen(\varphi)$.

In the following, the empty k-ucCST and the empty k-dcCST will be denoted, respectively, by k-ucCST_Empty_set and k-dcCST_Empty_set.

Other classes of sets can be encoded with k-CSTs. For instance, upward and downward closed sets of tuples of strings with respect to the sub-string relation can be efficiently encoded with k-CSTs. These sets are particularly interesting as they are manipulated by algorithms verifying properties on *Lossy FIFO channels*, a model where finite components communicate through unreliable channels.

To simplify the presentation we will often omit the encoding strategy f in the following. In those cases, we will consider the encoding strategy f defined such that $f(i) = i$ for all i. In those cases, the generalisation to any encoding function is straightforward.

The following results explain the advantages of using k-sharing trees for representing formulas.

Lemma 11 Let φ be an $\mathcal{U}_k$-formula ($\mathcal{D}_k$-formula) in disjunctive form with d disjuncts, $\mathbf{S}_\varphi$ an irredundant k-ucCST (k-dcCST) and f an encoding function such that $cones(\mathbf{S}_\varphi)_f = [\![\varphi]\!]$. Then, the number of nodes in $\mathbf{S}_\varphi$ is bounded by $((d+1) \cdot k) + 1$.

Proof. Let us consider that φ is an $\mathcal{U}_k$-formula. Suppose that φ is in normal form (there is no redundant disjunct) and suppose that φ is composed of d disjuncts, i.e. $\varphi \equiv \varphi_1 \vee \ldots \vee \varphi_d$ where $\varphi_i \equiv x_1 \geq c_{i1} \wedge \ldots \wedge x_k \geq c_{ik}$.

Notice that $(k \cdot (d+1)) + 1$ is the number of nodes that we have when there is no sharing in $\mathbf{S}_\varphi$ (exactly one path passes in each nodes (except for the *root* node) and there is one end node for each path). So, starting from the pre-k-ucCST $\mathbf{S}$ encoding φ without sharing, we apply Algorithm 27 to construct a pre-k-ucCST that satisfies condition 5 of Definition 42. Let us prove that the resulting pre-k-ucCST contains at most $(k \cdot (d+1)) + 1$ nodes. To achieve this goal, we prove that when Algorithm 27 finishes to treat a node (in the loop in line 1), the number of useful nodes is bounded by $(k \cdot (d+1)) + 1$ and all the nodes have at most one useful predecessor. An useful node is a node that is either the root node with at least one successor, an end node with at least one predecessor, or another node with at least one predecessor and one successor.

We exhibit a reasoning on the number i of nodes already treated by the loop of line 1 (in Algorithm 27) when applied on the pre-k-ucCST $\mathbf{S}$ without sharing where paths correspond to the tuples $\langle c_{i1}, \ldots, c_{ik} \rangle$ where i is such that $1 \leq i \leq d$. First, notice that only useful nodes are treated in loop of line 1 since before treating the nodes of the jth layer the useless nodes of that layer are removed by the loop of line 2 (see Algorithm 27).

Base case: i = 0. This case is trivially verified.

Induction step: i = k. Suppose that after treating k nodes, the induction hypothesis still holds. Let us prove that it is also the case after treating $k+1$ nodes. Let n be the $k+1th$ node treated by the loop of line 1 (see Algorithm 27). For each possible value c labelling the successors of n, the loop does the following. Let $\{n_1, \ldots, n_l\}$ be the set of successors of n labelled by c, i.e. $val^{\mathbf{S}}(n_i) = c$ for $1 \leq i \leq l$. The algorithm constructs a new node n' labelled by c such that $succ^{\mathbf{S}}(n') = succ^{\mathbf{S}}(n_1) \cup \ldots \cup succ^{\mathbf{S}}(n_l)$. After creating such nodes for each possible value c labelling the successors of n, it replaces $n_1, \ldots, n_l$ by n' in the set of successors

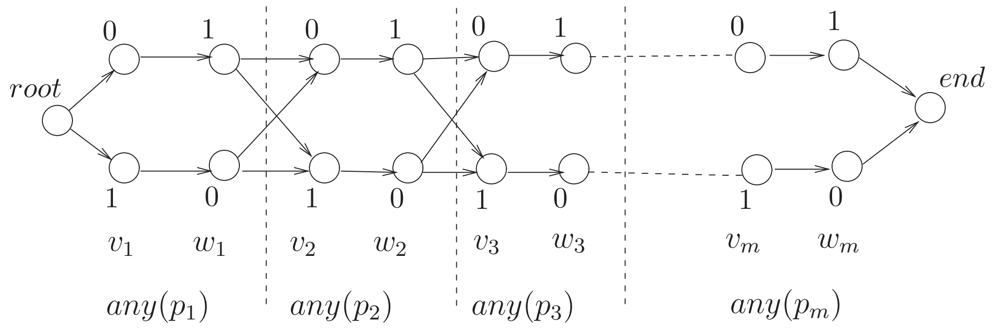

Figure 4.6: k-Sharing tree for a formula in disjunctive form.

of n. By induction hypothesis, n was the only useful predecessor of $n_1, \ldots, n_l$, hence those nodes become useless (their useless predecessors was removed from the **S** before treating the layer of n). As a consequence, the only useful predecessor of $n'' \in succ^{\mathbf{S}}(n_i)$ $(1 \leq i \leq l)$ becomes n'. Moreover, for each new successor of n, at least one old successor becomes useless. We conclude that after treating n the number of useful nodes has not growth and all the nodes have at most one useful predecessor.

We conclude that after applying Algorithm 27 on the pre-k-ucCST **S** without sharing where paths corresponds to generators of φ, we obtain a pre-k-ucCST **S**' with at most $(k\cdot(d+1))+1$ nodes. Since Algorithm 28 does not add new nodes, we conclude that the irredundant k-ucCST $\mathbf{S}_\varphi$ that corresponds to φ, obtained by applying Algorithm 28 on **S**', has at most $(k\cdot(d+1))+1$ nodes. □

As the number of edges in a k-ucCST (k-dcCST) is bounded by n^2 where n is the number of nodes, the following theorem holds.

Theorem 11 Let φ be an $\mathcal{U}_k$-formula ($\mathcal{D}_k$-formula) in disjunctive form with d disjuncts, $\mathbf{S}_\varphi$ an irredundant k-ucCST (k-dcCST) and f an encoding function such that $cones(\mathbf{S}_\varphi)_f = [\![\varphi]\!]$. Then, the size $\mathbf{S}_\varphi$ is bounded by $((k\cdot(d+1))+1)^2 + (k\cdot(d+1))+1$.

Proof. This is a direct implication of Lemma 11. □

From Theorem 11, we can see that all the formulas φ for which IDD and NDD fail to encode them efficiently are encoded by a k-ucCST (k-dcCST) with a size polynomial in size of φ. The following theorem states that there exist k-CSTs that are particularly concise representations of formulas.

Theorem 12 There exists an $\mathcal{U}$-formula ($\mathcal{D}$-formula) in disjunctive normal form φ such that the corresponding k-ucCST (k-dcCST) $\mathbf{S}_\varphi$ has a size logarithmic in the size of φ.

Proof. Let us first focus on $\mathcal{U}$-formulas.

Consider now the $\mathcal{U}$-formula over the set of variables $\{v_1, \ldots, v_n, w_1, \ldots, w_n\}$

$$\Phi = \bigwedge_{i=1}^{m} (v_i \geq 1 \wedge w_i \geq 0) \vee (v_i \geq 0 \wedge w_i \geq 1).$$

The corresponding disjunctive formulas φ (obtained by distributing $\wedge$ over $\vee$) is

$$\bigvee_{\{\mathbf{c}\in\{0,1\}^m\}} \bigwedge_{i=1}^{m} (v_i \geq c_i \wedge w_i \geq 1 - c_i)$$

This formula has 2^m disjuncts while being in normal form. In contrast, as shown in Figure 4.6, the k-ucCST $\mathbf{S}_\varphi$ has a size $10m + 2 = \mathcal{O}(m)$, i.e. logarithmic in the size of φ and polynomial in the size of Φ. In fact, each layer has two nodes and at most four arcs.

Replacing $\geq$ by $\leq$ in Φ, we obtain

$$\Phi' = \bigwedge_{i=1}^{m} (v_i \leq 1 \wedge w_i \leq 0) \vee (v_i \leq 0 \wedge w_i \leq 1).$$

Making a similar reasoning than on Φ, we conclude that the k-dcCST $\mathbf{S}_{\Phi'}$ is logarithmic in the size of the formula corresponding to the disjunctive form of Φ'. □

As for k-sharing trees, where there is a *canonical form* for finite sets of tuples, irredundant k-CSTs are a canonical form for closed sets in $(D^k, \preccurlyeq)$ when $(D^k, \mathcal{E}^k, \preccurlyeq)$ is well-formed.

Theorem 13 Let $(D^k, \mathcal{E}^k, \preccurlyeq)$ be well formed, S be a closed set of tuples in $(D^k, \preccurlyeq)$ and let $\mathbf{S}$ and $\mathbf{S}'$ be two irredundant k-CSTs . Then, we have $cones(\mathbf{S}) = S$ and $cones(\mathbf{S}') = S$ if and only if the k-CSTs $\mathbf{S}$ and $\mathbf{S}'$ are isomorphic.

Proof. From Proposition 4, as $cones(\mathbf{S}) = cones(\mathbf{S}')$ and $\mathbf{S}$, $\mathbf{S}'$ are irredundant, $elem(\mathbf{S})$ and $elem(\mathbf{S}')$ are minimal sets of generators of $cones(\mathbf{S})$ (and hence $cones(\mathbf{S}')$), then we have from Proposition 3 that $elem(\mathbf{S}) = elem(\mathbf{S}')$ since $\preccurlyeq$ is well-suited. It is proved in [ZC95] that two such k-sharing trees are isomorphic. □

From Theorem 13, we obtain that k-ucCSTs and k-dcCSTs are canonical forms for upward and downward closed sets of tuples in $\mathbb{N}^k$.

Corollary 9 Let Φ be a $\mathcal{U}$-formula ($\mathcal{D}$-formula) and let $\mathbf{S}$ and $\mathbf{S}'$ be two irredundant k-ucCSTs (k-dcCSTs). Then, we have $cones(\mathbf{S}) = [\![\Phi]\!]$ and $cones(\mathbf{S}') = [\![\Phi]\!]$ if and only if the k-ucCSTs (k-dcCSTs) $\mathbf{S}$ and $\mathbf{S}'$ are isomorphic.

In the sequel we will assume that all the k-CSTs have k layers.

4.5 Symbolic Operations for $\mathcal{U}$-formulas and $\mathcal{D}$-formulas

In this section we show how operations on the disjunctive formulas φ and ψ can be defined symbolically on their representations given by the k-ucCSTs (k-dcCSTs) $\mathbf{S}_\varphi$ and $\mathbf{S}_\psi$. We use the term *symbolically* because the algorithms that we propose work directly on the graphs $\mathbf{S}_\varphi$ and $\mathbf{S}_\psi$ and not by *enumerating* the tuples that they represent (or even the minimal (maximal) elements that they represent). All the results are given on k-CST but are directly extended to the special cases of k-ucCSTs and k-dcCSTs. In the sequel we will show that the complexity of the operations strongly depends on the form of the k-CSTs we consider. Specifically, we will distinguish between operations that require all arguments to be *irredundant* (abbreviated as IRR) k-CSTs, and operations that relax this condition and accept *generic* (abbreviated as GEN) k-CSTs in input. Table 4.2 summarizes the complexity results presented in this section for k-ucCSTs. A table that summarizes the results for k-dcCST is not given as the results for k-dcCST are exactly the same than for k-ucCST. So such a table can be directly inferred from Table 4.2.

In the following, we assume that $(D^k, \mathcal{E}^k, \preccurlyeq)$ is well-formed. We also assume that the considered partial order $\sqsubseteq$ is such that there is at least two elements $e, e' \in D \cup E$ with $e \neq e'$

and $e \sqsubseteq e'$. Notice that if it is not the case, we obtain k-sharing trees, for which the results of the complexity problems considered in the following are well-known (see Table 4.1). Moreover, we consider that

- the partial order $\sqsubseteq$ on the elements in $D \cup E$ is decidable in polynomial time in the size of the encodings of the elements in $D \cup E$ (as in the case of integers and strings); and
- There exists an algorithm to compute the operation $\cap_e$ on sets of elements in $D \cup E$.

The operation $\cap_e$ is defined as follows. Given finite generator sets $L, L' \subseteq (D \cup E)$ of the closed sets S and S' in $(D, \sqsubseteq)$ respectively, we have $L \cap_e L' = L''$ with $L'' \subseteq (D \cup E)$ such that L'' is a finite generator set of the set $\{\mathbf{t} \in D | \mathbf{t} \in S \text{ and } \mathbf{t} \in S'\}$. In the case of upward (downward) closed sets of natural numbers represented by their unique generators $\{v_1\}$ and $\{v_2\}$, we may define $\cap_e$ as follows: $\{v_1\} \cap_e \{v_2\} = \{max(v_1, v_2)\}$ ($\{v_1\} \cap_e \{v_2\} = \{min(v_1, v_2)\}$).

All the algorithms, except the first one (to decide the emptiness), presented in this section only work on non-empty k-CSTs. Actually, we can extend the algorithms by adding a special treatment for the case where the k-CST describing the empty set is given as parameter. For instance, the union between a k-CST $\mathbf{S}$ and the empty k-CST always returns $\mathbf{S}$. So, when computing the union between two k-CSTs, it suffices to test if one of the two k-CSTs represents the empty set. If it is the case, the algorithm returns the other k-CST.

Let us start from the simplest operations on closed sets on D^k.

Operation	**Abstract property of the operation**	**Argument type** (Irredundant/Generic)	**Complexity**	
			Problem	**Procedure**
Emptiness	$\mathsf{CST\_Empty}(\mathbf{S}) = \mathit{true}$ iff $\mathit{cones}(\mathbf{S}) = \emptyset$	GEN	constant	constant
Satisfiability	$\mathsf{CST\_Sat}(\mathbf{t}, \mathbf{S}) = \mathit{true}$ iff $\mathbf{t} \in \mathit{cones}(\mathbf{S})$	GEN	linear	linear
Subsumption	$\mathsf{CST\_Subsumption}(\mathbf{T}, \mathbf{S}) = \mathit{true}$ iff $\mathit{cones}(\mathbf{T}) \subseteq \mathit{cones}(\mathbf{S})$	IRR $\subseteq$ IRR	coNP	exponential
		GEN $\subseteq$ GEN	coNP	exponential
Equivalence	$\mathsf{CST\_Equiv}(\mathbf{S}, \mathbf{T})$ iff $\mathit{cones}(\mathbf{S}) = \mathit{cones}(\mathbf{T})$	IRR $\equiv$ IRR	linear	linear
		GEN $\equiv$ GEN	coNP	exponential
Normal form	$\mathsf{CST\_NormalForm}(\mathbf{S}) = \mathbf{T}$ s.t. $\mathit{elem}(\mathbf{T}) = \{\mathbf{m} \in \mathit{elem}(\mathbf{S}) \mid \nexists \mathbf{m}' \in \mathit{elem}(\mathbf{S}) : \mathbf{m}' \prec \mathbf{m}\}$	GEN	coNP	exponential
Disjunction	$\mathsf{CST\_Union}(\mathbf{S}, \mathbf{T}) = U$ s.t. $\mathit{cones}(U) = \mathit{cones}(\mathbf{S}) \cup \mathit{cones}(\mathbf{T})$	GEN $\cup$ GEN $=$ GEN	polynomial	polynomial
		IRR $\cup$ IRR $=$ IRR	coNP	exponential
Conjunction	$\mathsf{CST\_Intersection}(\mathbf{S}, \mathbf{T}) = \mathbf{I}$ s.t. $\mathit{cones}(I) = \mathit{cones}(\mathbf{S}) \cap \mathit{cones}(\mathbf{T})$	GEN $\cap$ GEN $=$ GEN	coNP	exponential
		IRR $\cap$ IRR $=$ IRR	coNP	exponential
Minimization	$\mathsf{CST\_Minimization}(\mathbf{S}, \mathbf{T}) = \mathbf{D}$ s.t. $\mathit{elem}(\mathbf{D}) = \{\mathbf{m} \in \mathit{elem}(\mathbf{S}) \mid \mathbf{m} \notin \mathit{cones}(\mathbf{T})\}$	GEN $\div$ GEN $=$ GEN	coNP	exponential
		IRR $\div$ IRR $=$ IRR	coNP	exponential

Table 4.2: Complexity of the operations on k-ucCSTs.

4.5.1 Emptiness Test

First of all, let us note that emptiness on k-CST can be checked in constant time. In fact, it is sufficient to test if the k-CST contains only the *root* and *end* nodes.

4.5.2 Satisfiability Test

Given a tuple $\mathbf{t} \in D^k$ and a k-CST $\mathbf{S}$, we focus in this section on the problem that consists to decide if $\mathbf{t} \in cones(\mathbf{S})$. This test for $\mathbf{t}$ on the k-CST $\mathbf{S}$ has a cost *polynomial* in the size of $\mathbf{S}$, i.e., possibly logarithmic in the size of the set of paths in $\mathbf{S}$. In fact, the following theorem holds.

Theorem 14 Let $\mathbf{S}$ be a k-CST and $\mathbf{t}$ be a vector of length k. We can check if there exists $\mathbf{t}' \in elem(\mathbf{S})$ such that $\mathbf{t}' \preccurlyeq \mathbf{t}$, namely $\mathbf{t} \in cones(\mathbf{S})$, in time polynomial in the number of edges of $\mathbf{S}$.

Proof. We exhibit an algorithm that has time complexity that is polynomial in the size of $\mathbf{S}$. The algorithm is based on a *layer-by-layer* comparison of the labels of the nodes of $\mathbf{S}$ with the components of the vector: at a given layer, we mark all nodes that belong to a path (tuple) that is candidate to subsume the vector. Specifically, we first mark $root^{\mathbf{S}}$. Then, if a node m at the i-th layer of $\mathbf{S}$ is marked, we mark all successors n of m such that $val(n) \sqsubseteq t_{i+1}$. At the end of the procedure, $\mathbf{t} \in cones(\mathbf{S})$ holds if and only if the node $end^{\mathbf{S}}$ is marked (note: we assume that $t_{k+1} = \perp$ and $\perp \sqsubseteq \perp$). □

4.5.3 Subsumption

Given two k-CSTs $\mathbf{S}$ and $\mathbf{S}$', we focus in this section on the problem that consists to decide if $cones(\mathbf{S}) \subseteq cones(\mathbf{S}')$. Let us first remark the following property. In any *irredundant* k-CST the constraints associated with a given variable (i.e. to the nodes of the corresponding layer) are not required to represent disjoint sets of tuples.

For example, let us look at the k-ucCST $\mathbf{S}_{val(F)}$ of Figure 4.8 (forget the rest of the picture for the moment). The root node of this k-ucCST has two successor nodes. The first one is labelled by 2 and the second one is labelled by 1, i.e. they have different labels but, for instance, the tuple $\langle 2, 2, 2, 2, 2, 2\rangle$ is encoded by paths passing through the first node and by paths passing through the second node. Starting from the root node, if we try to construct a path in the k-ucCST that encodes a tuple $\mathbf{t}$ with $t_1 = 2$, we cannot deterministically choose one successor node of *root*: the tuple can be encoded by a path passing through the first node or/and by a path passing through the second one (or there does not exit a path that encodes the tuple).

In other words, we may look at a k-CST $\mathbf{S}$ as a sort of non-deterministic automata accepting the tuples of his *rich denotation*. Non-determinism is the reason why we often obtain a compact representation. Subsumption between two k-CSTs however has the complexity of *trace inclusion* for non-deterministic, acyclic automata. We first analyze the complexity of the problem for *irredundant* k-CSTs.

The case $IRR \subseteq IRR$. In this case we have the following result.

Theorem 15 The subsumption problem for two irredundant k-CSTs is coNP-complete in the sum of the size of the k-CSTs.

Proof. Let $\mathbf{S}_1$ and $\mathbf{S}_2$ be irredundant k-CSTs. The problem is checking whether or not $cones(\mathbf{S}_1) \subseteq cones(\mathbf{S}_2)$. We can decide the complement of the problem by guessing a tuple of length k and testing in polynomial time (cf. Theorem 14) that it is subsumed by $\mathbf{S}_1$ and not by $\mathbf{S}_2$, restricting our attention to the finite but potentially exponential set of tuples (as in the proof of Theorem 8, due to Lemma 10) composed of values that appear in $\mathbf{S}_1$.

To prove hardness, we use a reduction from the *validity* problem as in Theorem 7. More precisely, given a propositional formula $F = D_1 \vee \ldots \vee D_d$ in DNF, we define (polynomially) the k-CST $\mathbf{S}_{val(F)}$ that encodes all valuations that satisfy F. We reduce then validity to the subsumption test between $\mathbf{S}_{val(true)}$ and $\mathbf{S}_{val(F)}$.

Following the proof of Theorem 7, we first build (polynomially) the two $\mathcal{U}$-formulas $\Phi_{val(true)}$ and $\Phi_{val(F)}$ that encode the valuations that make *true* and F true, respectively (for theorem 7, we worked in $\mathbb{N}^k$ but only used the constant 0 and 1; here we work in any structure $(D \cup E)^{2m}$ but we may replace 0 and 1 by two distinct values v_1, $v_2 \in D \cup E$ with $v_1 \neq v_2$ and $v_1 \sqsubseteq v_2$). Consider the $\mathcal{U}$-formulas $\Phi_{val(D_i)}$ we used in the proof of Theorem 7 to represent all valuations that satisfy a disjunct D_i of F. The transformation of $\Phi_{val(D_i)}$ to a disjunctive formula gives a formula φ_i with a size possibly exponential in the size of $\Phi_{val(D_i)}$. However, the representation of φ_i using a covering sharing tree $\mathbf{S}_{\varphi_i}$ is polynomial in the size of the *original* formula $\Phi_{val(D_i)}$, i.e., possibly logarithmic in the $|\varphi_i|$. Indeed, we can build $\mathbf{S}_{\varphi_i}$ directly from $\Phi_{val(D_i)}$ (the formal proof is by induction on the structure of the formula). Following the previous observations, we know that the k-CST $\mathbf{T}_{val(true)}$ (with root r_t and terminal node e_t) that encodes $\Phi_{val(true)}$, and the k-CST $\mathbf{S}_{\varphi_i}$ (with root r_i and terminal node e_i) that encodes $\Phi_{val(D_i)}$ have sizes polynomial in F. It remains now to define the irredundant k-CST $\mathbf{S}_{val(F)}$. Note that, as discussed in Section 4.5.5, the *union* operation of Table 4.1 does not guarantee the result to be an irredundant k-CST. Thus, instead of using *union* we proceed as follows. By construction, every k-CST $\mathbf{S}_{\varphi_i}$ is irredundant since we encoded the value p and $\neg p$ (for any proposition p of F) by two incomparable pairs of values. Based on this fact, we first merge all $\mathbf{S}_{\varphi_i}$'s into a single k-CST $\mathbf{S}_{val(F)}$ with root r and terminal node e. As paths of two different k-CST $\mathbf{S}_{\varphi_i}$ and $\mathbf{S}_{\varphi_j}$ could be comparable, we add a special prefix to the paths of each k-CST. These prefixes will ensure the irredundant of the k-CST $\mathbf{S}_{val(F)}$.

The $2\lceil log_2(d)\rceil$ first layers after the root of $\mathbf{S}_{val(F)}$ are constructed as follows (where $\lceil\cdot\rceil$ denotes ceiling function). In the first layer, we put two nodes n_1 and n_2 such that $val^{\mathbf{S}_{val(F)}}(n_1) = v_1$, $val^{\mathbf{S}_{val(F)}}(n_2) = v_2$ and $succ^{\mathbf{S}_{val(F)}}(r) = \{n_1, n_2\}$. Then, we proceed in a top-down manner. For each node n of odd layers, we add one node n' in the next layer such that $succ^{\mathbf{S}_{val(F)}}(n) = n'$ and $val^{\mathbf{S}_{val(F)}}(n') = v_1$ if $val^{\mathbf{S}_{val(F)}}(n) = v_2$, otherwise we have $val^{\mathbf{S}_{val(F)}}(n') = v_2$. For each node n of even layers (except the last one), we add two nodes n'_1 and n'_2 in the next layer such that $succ^{\mathbf{S}_{val(F)}}(n) = \{n'_1, n'_2\}$, $val^{\mathbf{S}_{val(F)}}(n'_1) = v_1$ and $val^{\mathbf{S}_{val(F)}}(n'_2) = v_2$. Following this procedure, the constructed prefixes are incomparable between each others (the proof is by induction in the size of the prefixes), the last layer has at least d nodes, and they are constructed in a time polynomial in the number d of disjuncts, so in a time polynomial in the size of the formula F. Figure 4.7 shows the fourth layers constructed when we want to merge four k-CSTs.

Let $\{n_1, \ldots, , n_{\lceil log_2(d')\rceil}\}$ (with $d' = 2^{\lceil log_2(d)\rceil}$) be the nodes of the last layer encoding the prefixes. We merge all the k-CSTs S_{φ_i}'s by defining $succ^{\mathbf{S}_{val(F)}}(n_i) = succ^{\mathbf{S}_{\varphi_i}}(r_i)$ for all $1 \leq i \leq d$, and modifying the successors of nodes when the set of successors contain

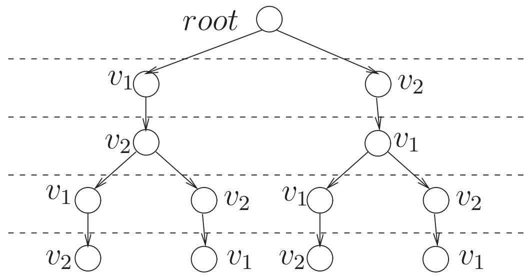

Figure 4.7: Prefixes constructed to merge four k-CSTs.

the terminal node. More precisely, when we have $succ^{\mathbf{S}_{\varphi_i}}(n) = \{e_i\}$ for some $1 \leq i \leq d$ we define $succ^{\mathbf{S}_{val(F)}}(n) = \{e\}$. To obtain a k-CST, we finally remove the nodes without successors (created during the construction of prefixes, if d is not a power of 2), and then apply Algorithm 28 (shown in Appendix) to ensure condition 6 of Definition 42. Clearly, the resulting k-CST is *irredundant* and has polynomial size in the size of F.

Finally, since the added prefixes should not influence the comparison of the paths of $\mathbf{S}_{val(true)}$ and $\mathbf{S}_{val(F)}$, we add the prefix of size $2\lceil log_2(n)\rceil$ only composed by the value v_2 to the paths of $\mathbf{S}_{val(true)}$.

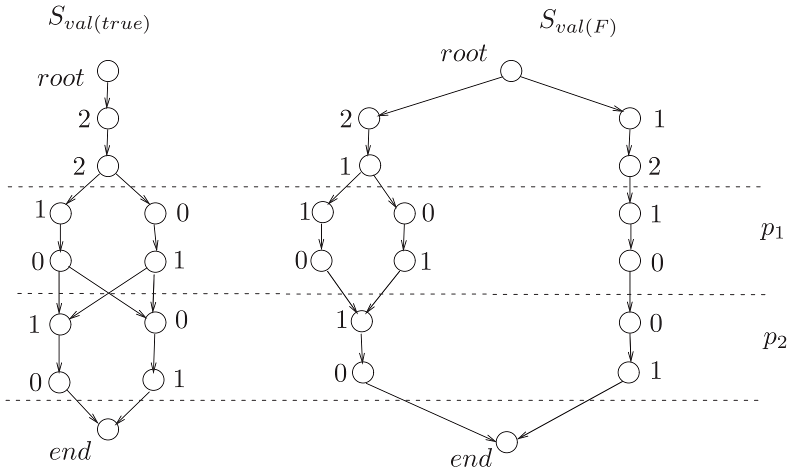

Figure 4.8: The validity of $F = p_2 \vee (p_1 \wedge \neg p_2)$ is reduced to $cones(\mathbf{S}_{val(true)}) \subseteq cones(\mathbf{S}_{val(F)})$.

We give an example of the reduction in Figure 4.8 using k-ucCSTs and where the values 1 and 2 are used to encode the prefixes. Based on this construction, we obtain that

$$F \text{ is valid iff } cones(\mathbf{S}_{val(true)}) \subseteq cones(\mathbf{S}_{val(F)}).$$

Since the reduction is *polynomial*, we obtain that the subsumption problem for k-CSTs is coNP-complete. □

The case $GEN \subseteq GEN$. From Theorem 15, we obtain the following corollary.

Corollary 10 The subsumption problem for two k-CSTs is coNP-complete in the size of the k-CSTs.

Proof. Given $\mathbf{S}_1$ and $\mathbf{S}_2$, we want to check $cones(\mathbf{S}_1) \subseteq cones(\mathbf{S}_2)$.

We can decide the complement of the problem by guessing a tuple of length k and testing (following Theorem 14, in linear time) that it is subsumed by $\mathbf{S}_1$ and not by $\mathbf{S}_2$. We restrict our attention to a finite but potentially exponential set of tuples as in the proof of Theorem 8 by only looking at tuples composed of values that appear in the k-CSTs.

Hardness follows from Theorem 15, since, the case $IRR \subseteq IRR$ is included in the case $GEN \subseteq GEN$. □

Algorithm 1: CST_Subsumption

Input: $\mathbf{S}, \mathbf{S}'$: k-CSTs
Result: *boolean*
begin
  $\mathbf{S}'' \leftarrow$ CST_Minimization($\mathbf{S}, \mathbf{S}'$);
  return CST_Empty($\mathbf{S}''$);
end

Algorithm for Subsumption The subsumption test can be performed symbolically via Algorithm 1, based on the procedures CST_Empty, that tests if the k-CST S'' is the k-CST that represents the empty set, and CST_Minimization that we will discuss in Section 4.5.7.

4.5.4 Equivalence Test and Reduction in Normal Form

Given two k-CSTs $\mathbf{S}$ and $\mathbf{S}'$, we first study the complexity of the equivalence test, i.e. the complexity of deciding if $cones(\mathbf{S}) = cones(\mathbf{S}')$. The equivalence test is easier than *subsumption*. This will be evident when considering comparisons of irredundant k-CSTs.

The case $IRR \equiv IRR$.

Theorem 16 Let $\mathbf{S}$ and $\mathbf{S}'$ be two *irredundant* k-CSTs. Then, equivalence of $\mathbf{S}$ and $\mathbf{S}'$ can be decided in *linear time*.

Proof. From Theorem 13, it follows that two irredundant k-CSTs are equivalent if and only if they are isomorphic. To check the latter condition, we can use a parallel top-down traversal. □

The case $GEN \equiv GEN$. The complexity of the equivalence test becomes harder, whenever one of the operands is not irredundant.

Theorem 17 Let $\mathbf{S}$ and $\mathbf{S}'$ be two k-CSTs. Then, equivalence of $\mathbf{S}$ and $\mathbf{S}'$ is coNP-*complete*.

Proof. We can decide the complement of the problem by guessing a tuple of length k and testing in polynomial time that it is subsumed by $\mathbf{S}$, but not by $\mathbf{S}'$, or (vice versa) that it is subsumed by $\mathbf{S}'$, but not by $\mathbf{S}$. We restrict again our attention to a finite but potentially exponential set of tuples as in the proof of Theorem 8 by only looking at tuples composed of values that appear in the k-CSTs.

To prove hardness, let us first notice that $cones(\mathbf{S}) \subseteq cones(\mathbf{S}')$ if and only if $cones(\mathbf{S}) \cup cones(\mathbf{S}') = cones(\mathbf{S}')$. Furthermore, notice that a simple way to compute a generic k-CST $\mathbf{T}$ with $cones(\mathbf{T}) = cones(\mathbf{S}) \cup cones(\mathbf{S}')$ is to take $\mathbf{T}$ as the *k-sharing tree* union of $\mathbf{S}$ and $\mathbf{S}'$, i.e., $\mathbf{T} = \mathsf{union}(\mathbf{S}, \mathbf{S}')$ (see [ZC95] for details on the algorithm). $\mathbf{T}$ will probably contain many redundant tuples, but this will not be a problem for our purposes. In fact, using the previous observation, and by the property of the union operation between k-sharing trees, we get a polynomial reduction from k-CSTs-subsumption (for generic k-CSTs) to k-CSTs-equivalence (for generic k-CSTs). From Corollary 10, it follows that checking equivalence between generic k-CSTs is coNP-hard. □

The previous result also applies to the case in which only one of the two k-CST is irredundant but not in the case where the two k-CSTs are irredundant as stated by Theorem 16.

Corollary 11 Let $\mathbf{S}$ be a generic k-CST, and $\mathbf{S}'$ be an irredundant k-CST. Checking $cones(\mathbf{S}) = cones(\mathbf{S}')$ is a coNP-*complete* problem.

Proof. We use the same proof as for Theorem 17. □

As a consequence, we have the following result.

Corollary 12 Given a k-CST $\mathbf{S}$, there is no *polynomial* algorithm that computes the irredundant k-CST $\mathbf{S}'$ such that $cones(\mathbf{S}) = cones(\mathbf{S}')$ (unless P= NP).

Proof. Per absurdum, suppose we can use our algorithm as a polynomial-time oracle for reduction in normal form. Given the k-CSTs $\mathbf{S}$ and the irredundant k-CST $\mathbf{S}'$, we first use the *oracle* to compute in polynomial time the irredundant k-CST $\mathbf{T}$ such that $cones(\mathbf{S}) = cones(\mathbf{T})$. Then, we use Theorem 16 to decide in polynomial time if $cones(\mathbf{T}) = cones(\mathbf{S}')$. This way we have built a *polynomial* time algorithm that decides the problem of Corollary 11. This gives us a contradiction. □

Algorithm for normalization Algorithm 3 computes the normal form of a given k-CST $\mathbf{S}$ by first computing the k-CST **Sub** containing all the paths of $\mathbf{S}$ (strictly) subsumed by a distinct other path of $\mathbf{S}$. This is done by means of a parallel traversal defined in the procedure Subsumed_Paths_Of_S (Algorithm 2). As any path is subsumed by itself, we ensure that the parallel traversal compares only distinct paths using the flag *distinct*. *distinct* is put to *true* when the parallel traversal passes through two distinct nodes at some layer i. When the parallel traversal reaches the *end* node and *distinct* is equal to false, then the parallel traversal passed by the same nodes for all the layers of $\mathbf{S}$, and hence it considers the same path in the two parallel traversal. Otherwise, if *distinct* is equal to true when the parallel traversal reaches *end*, there is at least one layer where the parallel traversal passed through two different nodes, and hence it considers two distinct paths.

The k-CST encoding the set of paths of $\mathbf{S}$ subsumed by another (distinct) paths of $\mathbf{S}$ is computed recursively. The procedure Subsumed_Paths_Of_S($n_1, n_2, distinct$) returns a node rooting a sub-tree containing the sub-paths rooted by n_1 that are subsumed by sub-paths rooted by n_2. If $distinct = true$, this means that the parallel traversal already consider two distinct nodes in some layer and is guaranteed to consider distinct paths. So, the node returned

Algorithm 2: Subsumed_Paths_Of_S

```
Input: n_S : node, n'_S : node, distinct : boolean
Result: node
begin
    newdistinct ← distinct ;
    if val^S(n_S) = ⊥ then
        if distinct = true then
            Sol ← new node such that val^Sub(Sol) = ⊥ and succ^Sub(Sol) = ∅ ;
        else
            Sol ← null;
    else
        Sol ← new node s.t. val^Sub(Sol) = val^S(n_S) ;
        foreach s_S ∈ succ^S(n_S), s'_S ∈ succ^S(n'_S) do
            if s_S ≠ s'_S then
                newdistinct ← true;
            if val^S(s_S) ≥_e val^S(s'_S) then
                Son ← memo?(s_S, s'_S, newdistinct);
                if Son = notmemorized then
                    Son ← Subsumed_Paths_Of_S(s_S, s'_S, newdistinct);
                if Son ≠ null then
                    succ^Sub(Sol) ← succ^Sub(Sol) ∪ {Son};
        if succ^Sub(Sol) = ∅ then
            Sol ← null;
    memo!(n_S, n'_S, Sol, newdistinct);
    return Sol;
end
```

Algorithm 3: CST_NormalForm

```
Input: S: k-CST
Result: k-CST
begin
    Let Sub be the k-CST built as follows ;
    root^Sub ← Subsumed_Paths_Of_S(root^S, root^S, false);
    Adjust_Condition5(Sub);
    Adjust_Condition6(Sub);
    return minus(S, Sub);
end
```

by Subsumed_Paths_Of_S could root a sub-tree containing sub-paths that are sub-paths of both n_1 and n_2. In the case where $distinct = false$, those paths do not appear in the set of sub-paths rooted by the resulting node of the function.

Since Subsumed_Paths_Of_S can be called several times with the same parameters, the results of the application of Subsumed_Paths_Of_S are memorized into a cache. memo! is used to cache intermediate results, i.e. memo!$(n_1, n_2, Sol, newdistinct)$ is used to caches the node Sol, the resulting node of Subsumed_Paths_Of_S when applied on n_1, n_2 and $newdistinct$. The procedure memo? retrieves cached information, i.e . memo?$(s_{\mathbf{S}}, s'_{\mathbf{S}}, newdistinct)$ returns the node Sol that have been cached by applying memo!$(n_1, n_2, Sol, newdistinct)$, or returns the $notmemorized$ value if no information have ever been cached. Using memo! and memo?, we call the function Subsumed_Paths_Of_S at most once for each pair of nodes. Indeed, calling twice the function for the nodes n_1 and n_2 with the same value for $distinct$ will return the same node (rooting a sub-tree composed of the sub-paths from n_1 subsumed by sub-paths from n_2). Instead, before calling Subsumed_Paths_Of_S$(n_1, n_2, newdistinct)$ we first call memo?$(n_1, n_2, newdistinct)$. If the cache contains the node that we need (that would be returned by Subsumed_Paths_Of_S if we called the method), then we directly obtain the node that would be computed by Subsumed_Paths_Of_S. Otherwise, if the cache returns $notmemorized$, the cache does not contain the node that we need, i.e. the node has not been already computed, and we compute it by applying Subsumed_Paths_Of_S.

As a consequence, Subsumed_Paths_Of_S has polynomial complexity. Finally, we use the k-sharing tree difference operation minus to removes all the paths of $\mathbf{S}$ that are also in **Sub** (see [ZC95] for details on the algorithm used to compute minus). To adjust the conditions 5 and 6 (that may be violated after running the procedure Subsumed_Paths_Of_S), we need to apply Algorithm 27 and Algorithm 28 of Appendix .1. As the cost of adjusting condition 5 may be exponential, CST_NormalForm has a worst case exponential complexity. However, we will see in Section 5.5 that in practice we are most of the time very far from the worst case.

4.5.5 Disjunction

Given two k-CSTs $\mathbf{S}$ and $\mathbf{S}'$, we focus in this section on the problem of computing the k-CST $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathbf{S}) \cup cones(\mathbf{S}')$. Let us first consider the case in which the resulting k-CST does not need not to be irredundant.

The case $GEN \cup GEN = GEN$. As already observed in the proof of Theorem 17, in this case we have an algorithm that works in polynomial time.

Theorem 18 Let $\mathbf{S}$ and $\mathbf{S}'$ be two k-CSTs. Then, we can compute a k-CST $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathbf{S}) \cup cones(\mathbf{S}')$ by resorting to the operation of union of k-sharing trees $\mathbf{T} = \mathsf{union}(\mathbf{S}, \mathbf{S}')$, polynomial in the size of $\mathbf{S}$ and $\mathbf{S}'$ (see Table 4.1).

The case $IRR \cup IRR = IRR$. The problem becomes *harder* if we require the resulting k-CST to be *irredundant*.

Theorem 19 Given two irredundant k-CSTs $\mathbf{S}$ and $\mathbf{S}'$, there is no *polynomial* algorithm that computes the irredundant k-CST $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathbf{S}) \cup cones(\mathbf{S}')$ (unless P= NP).

Proof. Per absurdum, suppose that we can use our algorithm as a polynomial-time oracle for computing the union of irredundant k-CSTs. Let us note that $cones(\mathbf{S}) \subseteq cones(\mathbf{S}')$ holds if and only if $cones(\mathbf{S}) \cup cones(\mathbf{S}') = cones(\mathbf{S}')$. In order to decide the subsumption problem $cones(\mathbf{S}) \subseteq cones(\mathbf{S}')$, we can first invoke the oracle to compute the irredundant k-CST $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathbf{S}) \cup cones(\mathbf{S}')$. Then, we use Theorem 16 to decide in polynomial time if $cones(\mathbf{T}) = cones(\mathbf{S}')$. This way we built a *polynomial* time algorithm to decide the *subsumption* problem for irredundant k-CSTs. This gives us the contradiction. □
The simplest way to compute disjunction is to first apply the polynomial algorithm for generic k-CSTs, and then use Algorithm 3 to get the resulting k-CST in normal form.

4.5.6 Conjunction

Given two k-CSTs $\mathbf{S}$ and $\mathbf{S}'$, we focus in this section on the problem of computing the k-CST $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathbf{S}) \cap cones(\mathbf{S}')$. Let us first study the case in which the operands are irredundant k-CSTs.

The case $IRR \cap IRR = IRR$. The following result shows that the intersection is hard when the result is required to be irredundant.

Theorem 20 Given two irredundant k-CSTs $\mathbf{S}$ and $\mathbf{S}'$, there is no *polynomial* algorithm that computes the irredundant k-CST $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathbf{S}) \cap cones(\mathbf{S}')$ (unless $\mathsf{P}=\mathsf{NP}$).

Proof. Per absurdum, suppose that we can use our algorithm as a polynomial-time oracle for computing the intersection of irredundant k-CSTs. Let us note that $cones(\mathbf{S}) \subseteq cones(\mathbf{S}')$ holds if and only if $cones(\mathbf{S}) \cap cones(\mathbf{S}') = cones(\mathbf{S})$. In order to decide the subsumption problem $cones(\mathbf{S}) \subseteq cones(\mathbf{S}')$, we can first invoke the oracle to compute the irredundant k-CST $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathbf{S}) \cap cones(\mathbf{S}')$. Then, we use Theorem 16 to decide in polynomial time if $cones(\mathbf{T}) = cones(\mathbf{S})$. This way we built a *polynomial* time algorithm to decide the *subsumption* problem for irredundant k-CSTs. This gives us the contradiction. □

The case $GEN \cap GEN = GEN$.

Theorem 21 Given two k-CSTs $\mathbf{S}$ and $\mathbf{S}'$, there is no *polynomial* algorithm that computes the k-CST $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathbf{S}) \cap cones(\mathbf{S}')$ (unless $\mathsf{P}=\mathsf{NP}$).

Proof. Following [Joh90], we exhibit a polynomial *Turing* reduction from *validity*. Specifically, we build an OTM (Oracle Turing Machine) that uses *intersection* as an oracle to decide in polynomial time *validity*. To prove the validity of a propositional formula F in DNF, we first construct the k-CSTs $\mathbf{S}_{val(F)}$ and $\mathbf{S}_{val(true)}$, as shown in the proof of Theorem 15. This step can be done in *polynomial* time. By construction, we obtain then that F is *valid* if and only if $cones(\mathbf{S}_{val(true)}) \subseteq cones(\mathbf{S}_{val(F)})$, i.e., if and only if $cones(\mathbf{S}_{val(true)}) \subseteq cones(\mathbf{S}_{val(F)}) \cap cones(\mathbf{S}_{val(true)})$.

We apply then the oracle to compute the k-CST $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathbf{S}_{val(true)}) \cap cones(\mathbf{S}_{val(F)})$. The value v_2 of nodes in the extra layers in $cones(\mathbf{S}_{val(true)})$ are greater than the values of nodes in the extra layers in $cones(\mathbf{S}_{val(F)})$. It follows that the following holds. When computing the intersection of the cones induced by $\mathbf{S}_{val(true)}$ with those of $\mathbf{S}_{val(F)}$, the resulting cones will always have minimal points with the value v_2 in the dimensions corresponding to extra layers used in the construction of Theorem 15. By hypothesis, $\mathbf{T}$ may contain redundant tuples. So we cannot apply a direct equivalence test between $\mathbf{T}$ and $\mathbf{S}_{val(true)}$ (as shown in Theorem 17, this test is exponential for generic operands). However, note that $\mathbf{S}_{val(true)}$ is irredundant by construction. Thus, to check $cones(\mathbf{S}_{val(true)}) \subseteq cones(\mathbf{T})$ we simply have to check that *at least* all generators of $\mathbf{S}_{val(true)}$ occur in $\mathbf{T}$, i.e., we have to check that $elem(\mathbf{S}_{val(true)}) \subseteq elem(\mathbf{T})$. This is the containment test between the flat denotation of *k-sharing trees* that we know to be polynomial (see Table 4.1). Thus if we can construct the k-CST $\mathbf{T}$ such that $cones(\mathbf{S}) \cap cones(\mathbf{S}')$ in polynomial time, we have an algorithm to test the validity of propositional formula in DNF. This makes a contradiction (unless P= NP) with the fact that the validity problem for propositional formula in DNF is a coNP complete problem. □

Algorithm for Conjunction Let us now describe the algorithm that we propose to compute the intersection between k-CSTs.

The idea of the algorithm is the following. An element $\mathbf{t}$ is in the intersection of two k-CST $\mathbf{S}_1$ and $\mathbf{S}_2$ if there is $\mathbf{t}_1 \in elem(\mathbf{S}_1)$ and $\mathbf{t}_2 \in elem(\mathbf{S}_2)$ such that $\mathbf{t}_1 \preccurlyeq \mathbf{t}$ and $\mathbf{t}_2 \preccurlyeq \mathbf{t}$, i.e. for each dimension i we have $t_{1i} \sqsubseteq t_i$ and $t_{2i} \sqsubseteq t_i$. The algorithm constructs a k-CST $\mathbf{S}$ where nodes are pairs $\langle n_1, n_2 \rangle$ where n_1 is a node of $\mathbf{S}_1$ and n_2 is a node of $\mathbf{S}_2$. A node $\langle n_1, n_2 \rangle$ is a successor of the node $\langle n_1', n_2' \rangle$ if n_1 is a successor of n_1' and n_2 is a successor of n_2'. Hence, each path of $\mathbf{S}$ corresponds to a path of $\mathbf{S}_1$ and a path of $\mathbf{S}_2$. The value a node $\langle n_1, n_2 \rangle$ is a generator of the closed set corresponding to the intersection between the closed set generated by $val^{\mathbf{S}_1}(n_1)$ and the closed set generated by $val^{\mathbf{S}_2}(n_2)$. So, a path of $\mathbf{S}$ that corresponds to a path p_1 of $\mathbf{S}_1$ and a path p_2 of $\mathbf{S}_2$ encodes tuples that are encoded by both p_1 and p_2.

More precisely, the algorithm is the following. Given the k-CSTs $\mathbf{S}_1$ and $\mathbf{S}_2$, we compute $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathbf{S}_1) \cap cones(\mathbf{S}_2)$ as follows. We first define a pre-k-CST $\mathbf{P}$ with the following components:

1. $N^{\mathbf{P}} = \{root\} \cup N_1^{\mathbf{P}} \cup \cdots \cup N_k^{\mathbf{P}} \cup \{end\}$ with $N_i^{\mathbf{P}} = \bigcup_{n \in N_i^{\mathbf{S}_1}} \bigcup_{m \in N_i^{\mathbf{S}_2}} \{\langle n_1, m_1 \rangle, \ldots, \langle n_l, m_l \rangle\}$ such that $val^{\mathbf{S}_1}(n) \cap_e val^{\mathbf{S}_2}(m) = \{v_1, \ldots, v_l\}$ and $val^{\mathbf{P}}(\langle n_i, m_i \rangle) = v_i$ for all $1 \le i \le l$;
2. we set $succ^{\mathbf{P}}(root) = N_1^{\mathbf{P}}$, for all $(n, m) \in N_1 \cup \ldots N_k : succ^{\mathbf{P}}(\langle n, m \rangle) = \{\langle n_i', m_i' \rangle | n' \in succ^{\mathbf{S}_1}(n)$ and $m' \in succ^{\mathbf{S}_2}(m)\}$, and for all $(n, m) \in N_k^{\mathbf{P}} : succ^{\mathbf{P}}((n, m)) = \{end\}$.

We obtain the k-CST $\mathbf{T}$ from $\mathbf{P}$ by enforcing conditions (5-6) of Definition 42 with the algorithms in Appendix .1, Figure 27 and Figure 28. The algorithm to adjust condition 5 may have exponential time complexity in the size of the k-CST, whereas the time complexity to adjust condition 6 is polynomial. Note that $\mathbf{T}$ can contain redundancies, even if $\mathbf{S}$ and $\mathbf{S}'$ are irredundant. To get an irredundant k-CST, we have to transform $\mathbf{T}$ in normal form.

4.5.7 Minimization

Given the k-CSTs $\mathbf{S}$ and $\mathbf{S}'$, the operations of simplification of $\mathbf{S}$ according to $\mathbf{S}'$ (noted $\mathbf{S} \div \mathbf{S}'$) returns the k-CST $\mathbf{T}$ such that $elem(\mathbf{T}) = \{\mathbf{m} \in elem(\mathbf{S}) |\ \nexists \mathbf{m}' \in cones(\mathbf{S}') : \mathbf{m}' \preccurlyeq \mathbf{m}\}$.

As already explained in Section 4.3.5, this operation is used by verification algorithms presented in Chapter 5 as follows. Intuitively, $\mathbf{S}$ will represent a set of states to be treated and $\mathbf{S}'$ will represent the set of states already treated. As it is not necessary to treat twice a state, generators of $\mathbf{S}$ that are subsumed by a generator of $\mathbf{S}'$, i.e. that generates a closed set included into the closed set represented by $\mathbf{S}'$, are removed using the operation $\div$.

Theorem 22 Given the k-CSTs $\mathbf{S}$ and $\mathbf{S}'$, there is no *polynomial* algorithm, that computes the k-CST $\mathbf{S} \div \mathbf{S}'$ (unless P= NP).

Proof. We exhibit a polynomial *Turing* reduction from *validity*. Specifically, we build an OTM (Oracle Turing Machine) that uses $\div$ as an oracle to decide in polynomial time *validity*. We first construct the two k-CSTs $\mathbf{S}_{val(F)}$ and $\mathbf{S}_{val(true)}$, as shown in the proof of Theorem 15.

We apply then the oracle on $\mathbf{S}_{val(true)}$ and $\mathbf{S}_{val(F)}$ to compute the k-CST $\mathbf{T} = \mathbf{S}_{val(true)} \div \mathbf{S}_{val(F)}$. Finally we note that F is valid if and only if $\mathbf{T}$ is the empty k-CST, i.e the k-CST that represents the empty set. Indeed, if F is valid, then all the path of $\mathbf{S}_{val(true)}$ are subsumed by at least one path in $\mathbf{S}_{val(F)}$. Checking emptiness can be done in constant time. Thus, the resulting Turing reduction is polynomial and the coNP hardness for our search problem follows from the coNP completeness of *validity*. □

Algorithm for minimization Algorithm 5 computes the minimization operation for two k-CSTs $\mathbf{S}$ and $\mathbf{T}$. The procedure CST_Minimization works similarly to Algorithm 3. The function CST_Minimization first applies a procedure called Paths_Of_S_Subsumed_By_T(see Algorithm 4) that computes by means of a parallel traversal of $\mathbf{S}$ and $\mathbf{T}$ the pre-k-CST $\mathbf{Sub}$ where the paths are those of $\mathbf{S}$ that are subsumed by at least one path in $\mathbf{T}$.

That procedure works recursively. Paths_Of_S_Subsumed_By_T(n_1, n_2) returns a node rooting a subtree containing all the sub-path in the sub-tree rooted by the node n_1 (in $\mathbf{S}$) that are subsumed by a path of the sub-tree rooted by the node n_2 (in $\mathbf{T}$). This is achieved by a parallel traversal of the sub-trees rooted by n_1 and n_2.

Since Paths_Of_S_Subsumed_By_T can be called several times with the same parameters, the results of those calls are memorized into a cache. memo! is used to cache intermediate results, i.e. memo!(n_1, n_2, Sol) is used to caches the node Sol, the resulting node of Paths_Of_S_Subsumed_By_T when applied on n_1 and n_2. The procedure memo? retrieves cached information, i.e . memo?($s_{\mathbf{S}}, s'_{\mathbf{S}}$) returns the node Sol that have been cached by applying memo!(n_1, n_2, Sol), or returns the *notmemorized* value if no information have ever been cached. Using memo! and memo?, we call the function Paths_Of_S_Subsumed_By_T at most once for each pair of nodes. Indeed, calling twice the function for the nodes n_1 and n_2 will return the same node (rooting a sub-tree composed of the sub-paths from n_1 subsumed by sub-paths from n_2). Instead, before calling Paths_Of_S_Subsumed_By_T(n_1, n_2) we first call memo?(n_1, n_2). If the cache contains the node that we need, then we directly obtain the node that would be computed by Paths_Of_S_Subsumed_By_T. Otherwise, if the cache returns *notmemorized*, the cache does not contain the node that we need, i.e. the node has not been already computed, and we compute it by applying Paths_Of_S_Subsumed_By_T.

As a consequence, Paths_Of_S_Subsumed_By_T has polynomial complexity. Then, Algorithm 27 (Adjust_Condition5) and Algorithm 28 (Adjust_Condition6) are applied as the pre-k-CST $\mathbf{Sub}$ returned by Paths_Of_S_Subsumed_By_T that possibly does not satisfy condition 5 and 6 of Definition 42

Finally, the minus operation is applied on $\mathbf{S}$ and $\mathbf{Sub}$. This operation removes all the path from $\mathbf{S}$ that appear in $\mathbf{Sub}$ (see [ZC95] for details on the algorithm computing the minus operation on k-sharing trees). This leads to a k-CST containing all the paths of $\mathbf{S}$ that are not subsumed by a path of $\mathbf{T}$.

The complexity of CST_Minimization is exponential in the size of $\mathbf{S}$ and $\mathbf{T}$ because Paths_Of_S_Subsumed_By_T returns a pre-k-CST that must be (possibly) determinized (as it would not satisfy condition 5 of Definition 42).

Algorithm 4: Paths_Of_S_Subsumed_By_T

```
Input: n_S : node, n_T : node
Result: node
begin
    if val^S(n_S) = ⊥ then
        Sol ← new node such that val^Sub(Sol) = ⊥ and succ^Sub(Sol) = ∅;
    else
        Sol ← new node s.t. val^Sub(Sol) = val^S(n_S);
        foreach s_S ∈ succ^S(n_S), s_T ∈ succ^T(n_T) do
            if val^S(s_S) ≥_e val^T(s_T) then
                Son ← memo?(s_S, s_T);
                if Son = notmemorized then
                    Son ← Paths_Of_S_Subsumed_By_T(s_S, s_T);
                if Son ≠ null then
                    succ^Sub(Sol) ← succ^Sub(Sol) ∪ {Son};
        if succ^Sub(Sol) = ∅ then
            Sol ← null;
    memo!(n_S, n_T, Sol);
    return Sol;
end
```

4.5.8 Approximations using Simulation Relations

In the previous section we have proved several hardness results related to important problems defined on k-CSTs. In this section we introduce techniques to approximates difficult operations on k-CST, like removing redundancy, the minimization operation and subsumption test. More precisely, we will introduce *approximations* of the subsumption relation that can be tested more efficiently. The main idea is to exploit the *graph-structure* underlying k-CSTs, and in particular the possibility of defining *relations* between nodes of k-CSTs.

Technically, we will try to find sufficient conditions for subsumption and equivalence by using *simulation relations*. Let $\mathbf{S}_1$ and $\mathbf{S}_2$ be two k-CSTs. Note that, if $elem(\mathbf{S}_1) \subseteq elem(\mathbf{S}_2)$ then $cones(\mathbf{S}_1) \subseteq cones(\mathbf{S}_2)$. Besides giving a sufficient condition for checking subsumption, the previous fact suggests a possible strategy to reduce the cost of the complete test. We first compute $\mathbf{T} = \mathsf{minus}(\mathbf{S}_1, \mathbf{S}_2)$ (polynomial in the size of $\mathbf{S}_1, \mathbf{S}_2$) and then test $cones(\mathbf{T}) \subseteq cones(\mathbf{S}_2)$ on the (possibly) smaller k-CST $\mathbf{T}$.

In the rest of this section we will give more interesting polynomial-time *sufficient conditions* for the subsumption test, based on a notion of *simulation* between nodes of k-CSTs. We will

Algorithm 5: CST_Minimization

Input: $\mathbf{S}$: k-CST, $\mathbf{T}$: k-CST
Result: k-CST
begin
  Let **Sub** be the k-CST built as follows ;
  $root^{\mathbf{Sub}} \leftarrow$ Paths_Of_S_Subsumed_By_T($root^{\mathbf{S}}, root^{\mathbf{T}}$);
  Adjust_Condition5(**Sub**);
  Adjust_Condition6(**Sub**);
  return minus($\mathbf{S}$, **Sub**);
end

see that this notion of simulation is also useful to approximate both the reduction in normal form and the simplification operation on k-CSTs.

Given two nodes n and m of a k-CST $\mathbf{S}$ we are looking for a relation $\curvearrowright_{\mathcal{F}}$ such that: $n \curvearrowright_{\mathcal{F}} m$ implies $cones(\mathbf{S}_n) \subseteq cones(\mathbf{S}_m)$ where $\mathbf{S}_n$ and $\mathbf{S}_m$ are the sub-k-CSTs rooted, respectively, by n and m.

Definition 52 (Forward Simulation) Given two k-CSTs $\mathbf{S}$ and $\mathbf{T}$, let n be a node of the i-th layer of $\mathbf{S}$, and m be a node of the i-th layer of $\mathbf{T}$. We say that n is *forward simulated by* m, written $n \curvearrowright_{\mathcal{F}} m$, if $val^{\mathbf{T}}(m) \sqsubseteq val^{\mathbf{S}}(n)$ and for all the nodes $s \in succ^{\mathbf{S}}(n)$ there exists a node $t \in succ^{\mathbf{T}}(m)$ such that $s \curvearrowright_{\mathcal{F}} t$.

Let $father^{\mathbf{S}}(n) = \{\ m \mid n \in succ^{\mathbf{S}}(m)\ \}$ be the set of fathers of a node n at layer i of a k-CST $\mathbf{S}$ ($fathers^{\mathbf{S}}(n) \subseteq N_{i-1}$). We define the backward simulation as follows:

Definition 53 (Backward Simulation) Given two k-CSTs $\mathbf{S}$ and $\mathbf{T}$, let n be a node of the i-th layer of $\mathbf{S}$, and m be a node of the i-th layer of $\mathbf{T}$. We say that n is *backward simulated by* m, written $n \curvearrowright_{\mathcal{B}} m$, if $val^{\mathbf{T}}(m) \sqsubseteq val^{\mathbf{S}}(n)$ and for all the nodes $s \in fathers^{\mathbf{S}}(n)$ there exists a node $t \in fathers^{\mathbf{T}}(m)$ such that $s \curvearrowright_{\mathcal{B}} t$.

If we see the forward (backward) simulations as relations, a forward (backward) simulation on the nodes of $\mathbf{S}$ and $\mathbf{T}$ is the weakest forward simulation on the nodes of $\mathbf{S}$ and $\mathbf{T}$ if it includes all the other forward (backward) simulations on the nodes of $\mathbf{S}$ and $\mathbf{T}$.

The following result (taken from [HHK95]) shows that the previous simulations can be computed efficiently.

Theorem 23 (From [HHK95]) The weakest forward and backward simulation relations between the nodes of the k-CST $\mathbf{S}$ and the nodes of the k-CST $\mathbf{T}$ can be computed in $O(m \cdot n)$ where m is the sum of the number of nodes in $\mathbf{S}$ and in $\mathbf{T}$, and n is the sum of the number of edges in $\mathbf{S}$ and in $\mathbf{T}$.

Properties of the simulation

The following propositions relate subsumption and the simulation $\curvearrowright_{\mathcal{F}}$.

Lemma 12 Given the k-CSTs $\mathbf{S}$ and $\mathbf{T}$, let $\mathbf{S}_n$ and $\mathbf{T}_m$ be the sub-k-CSTs rooted at nodes n and m, respectively. If $n \curvearrowright_{\mathcal{F}} m$ then $cones(\mathbf{S}_n) \subseteq cones(\mathbf{S}_m)$.

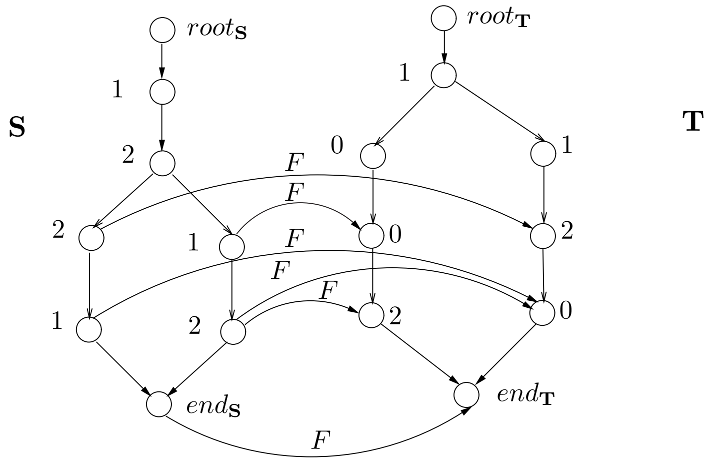

Figure 4.9: The forward simulation is incomplete with respect to subsumption.

Proof. We will show that, for every node n of $\mathbf{S}$ in layer $k-l$, for $0 \leq l \leq k$, the property is verified. We reason by induction on the value of l.

Base case: $l = 0$. Let us consider $n \in N^{\mathbf{S}}_{k-l}$ and $m \in N^{\mathbf{T}}_{k-l}$ such that $n \curvearrowright_F m$. By definition of k-CSTs, we have that $succ^{\mathbf{S}}(n) = \{end\}$, $succ^{\mathbf{T}}(m) = \{end\}$, $elem(\mathbf{S}_n) = \langle val^{\mathbf{S}}(n)\rangle$ and $elem(\mathbf{T}_m) = \langle val^{\mathbf{T}}(m)\rangle$; since by definition of $\curvearrowright_F$, we have that $val^{\mathbf{S}}(n) \geq_e val^{\mathbf{T}}(m)$, we also get that $cones(\mathbf{S}_n) = \{x \geq val^{\mathbf{S}}(n)\}$ is included in $cones(\mathbf{S}_m) = \{x \geq val^{\mathbf{S}}(m)\}$.

Induction step: $l > 0$. By induction hypothesis, for all nodes $n' \in N^{\mathbf{S}}_{k-l+1}$and $m' \in N^{\mathbf{T}}_{k-l+1}$, we know that if $n' \curvearrowright_F m'$ then $cones(\mathbf{S}_{n'}) \subseteq cones(\mathbf{S}_{m'})$. Let us now consider $n \in N^{\mathbf{S}}_{k-l}$ and $m \in N^{\mathbf{T}}_{k-l}$ with $n \curvearrowright_F m$. By definition of $\curvearrowright_F$, we know that for $s \in succ^{\mathbf{S}}(n)$ there exists $t \in succ^{\mathbf{T}}(m)$ such that $s \curvearrowright_F t$. We also know that $val^{\mathbf{S}}(n) \geq_e val^{\mathbf{T}}(m)$. Thus, we have that

$$elem(\mathbf{S}_n) = \{\{val^{\mathbf{S}}(n)\} \times elem(\mathbf{S}_{n'}) \mid n' \in succ^{\mathbf{S}}(n)\}$$

and

$$elem(\mathbf{T}_m) = \{\{val^{\mathbf{T}}(m)\} \times elem(\mathbf{T}_{m'}) \mid n' \in succ^{\mathbf{T}}(m)\}.$$

Thus, it follows that $cones(\mathbf{S}_n) \subseteq cones(\mathbf{T}_m)$ holds. □

4.5.9 Approximations using Simulation Relations

The converse does not hold (in accordance with the coNP hardness result for subsumption). As a counterexample, take the two k-ucCSTs in Figure 4.9. The curbed arrows represent the forward simulation relation between nodes of $\mathbf{S}$ and $\mathbf{T}$. Note that none of the nodes of layer 2 in $\mathbf{T}$ simulates the single node of $\mathbf{S}$ at the same layer. However, the denotation of $\mathbf{S}$ is contained in that of $\mathbf{T}$. In fact, $\langle 1,1,2,0\rangle \preccurlyeq \langle 1,2,2,1\rangle$ and $\langle 1,0,0,2\rangle \preccurlyeq \langle 1,2,1,2\rangle$.

As Lemma 12 can be reformulated for backward simulation, the following theorem holds.

Theorem 24 Let $root^{\mathbf{S}}, end^{\mathbf{S}}$ and $root^{\mathbf{T}}, end^{\mathbf{T}}$ be the root and terminal nodes of $\mathbf{S}$ and $\mathbf{T}$, respectively. If $root^{\mathbf{S}} \curvearrowright_F root^{\mathbf{T}}$ then $cones(\mathbf{S}) \subseteq cones(\mathbf{T})$. Symmetrically, if $end^{\mathbf{S}} \curvearrowright_B end^{\mathbf{T}}$ then $cones(\mathbf{S}) \subseteq cones(\mathbf{T})$.

The previous theorem gives us sufficient conditions for testing subsumption.

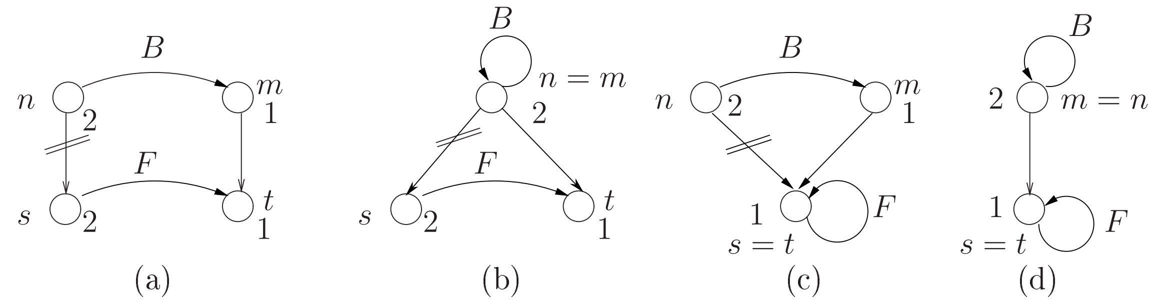

Figure 4.10: Use of the simulation relations to remove redundancies.

Use of simulations to remove redundancies

As for the subsumption test, the simulations we introduced in Section 4.5.8 can be used to approximate the exact normalization procedure and the exact simplification procedure. For this purpose, we introduce a rule that allows us to exploit the information given by the simulation relations in order to locally remove edges of a k-CST.

Definition 54 (Edge Removal) Given a k-CST $\mathbf{S}$ having two nodes $n, m \in N^{\mathbf{S}}$ with $n \curvearrowright^{B} m$ such that there exist $s \in succ^{\mathbf{S}}(n)$ and $t \in succ^{\mathbf{S}}(m)$ with $s \curvearrowright^{F} t$ and $s \neq t \vee n \neq m$, we define $remove(\mathbf{S}, \langle n, s \rangle)$ as the pre-k-CST with successor relation $succ'$ obtained from $\mathbf{S}$ by setting $succ'(n) = succ(n) \setminus \{s\}$.

Figure 4.10 shows the different cases that occur when we use simulation relations to remove redundancies. Figure 4.10(a) shows the case where the node n, m, s and t of Definition 54 are all distinct. As n is backward simulated by m, all the paths leading to n are subsumed by at least one path leading to m and as s is forward simulated by t we have that all the sub-paths starting from s are subsumed by at least one sub-path starting from t. We conclude that all the paths passing through the arc between n and s are subsumed by at least one path passing through the edge between m and t. So the edge between n and s can be removed.

Figure 4.10(b) and Figure 4.10(c) show similar situations but where, respectively, the nodes n and m are the same one and where the nodes s and t are the same one. Finally, 4.10(d) shows the case where n and m are the same node, and s and t are also the same nodes. In this case, simulation relations detect that the paths are subsumed by themselves, and hence does not detect real redundancies. In this last case, no edges can be removed.

The following proposition states the interest of the rule induced by Definition 54. Its correctness follows directly from the definition of *remove*.

Proposition 10 Given a k-CST $\mathbf{S}$ with the nodes m, n with $n \curvearrowright^{B} m$ such that there exists $s \in succ^{\mathbf{S}}(n)$ and $t \in succ^{\mathbf{S}}(m)$ $(s \neq t \vee n \neq m)$ with $s \curvearrowright^{F} t$, then

$$cones(\mathbf{S}) = cones(remove(\mathbf{S}, \langle n, s \rangle)).$$

A possible strategy to apply Definition 54 consists in the on-the-fly removal of edges during the computation of the simulation relations. Specifically, during a bottom-up traversal of the k-CST (assume the backward simulation is already computed), we first apply the rule of definition 54 to every pairs of edges between nodes of layers N_i and N_{i+1}. Removing edges

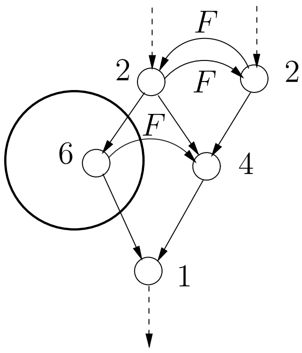

Figure 4.11: Violation of condition 6 after removing two edges and one node.

between nodes of these layers does not change the structure of the k-CST in the set of layers $S_1 = \{ N_j$ such that $j \leq i\}$ and the set of layers $S_2 = \{N_j$ such that $j \geq i+1 \}$. As a consequence, the backward simulation does not change for the nodes of the layers in S_1 and the forward simulation does not change for the nodes of the layers in S_2. This allows us to propagate the application of the rule bottom-up. The rule to remove edges is applied exhaustively at each step.

Note that, after the bottom-up traversal, the forward simulation is computed on the nodes of the resulting pre-k-CST, but the backward simulation could have changed. An example on a k-ucCST is given in Figure 4.12, where B and F represent backward and forward simulation, respectively. Since $3 \curvearrowright_B 2$ and $6 \curvearrowright_F 5$, we can remove the arc $3 \rightarrow 6$ during the bottom-up traversal, doing so we will add $\langle 6, 0\rangle$ to $\curvearrowright_B$. By doing a top-down traversal, some new edges could be removed again. As a consequence, we get better and better results iterating bottom-up and top-down traversals. Also note that the pre-k-CST $remove(\mathbf{S}, \langle n, s\rangle)$ may violate the condition 6 of the definition of k-sharing trees (Section 4.4.2). For instance, removing the node with label 6 from the k-ucCST in Figure 4.11 makes the two nodes with label 2 (belonging to vectors that are not in the subsumption relation) have the same set of successors. As already mentioned, condition 6 can be restored using the algorithm in Appendix .1, Figure 28, polynomial in the size of the k-CST.

To approximate the simplification procedure, we can introduce a definition similar to definition 54.

Definition 55 (Edge removal to approximate minimization) Given two k-CSTs $\mathbf{S}$ and $\mathbf{T}$ such that there are two nodes $n \in N^{\mathbf{S}}$ and $m \in N^{\mathbf{T}}$ with $s \in succ^{\mathbf{S}}(n)$ and $t \in succ^{\mathbf{S}}(m)$ such that with $n \curvearrowright_B m$ and $s \curvearrowright_F t$, we define $remove(\mathbf{S}, \langle n, s\rangle)$ as the pre-k-CST with successor relation $succ'$ obtained from $\mathbf{S}$ by setting $succ'(n) = succ(n) \setminus \{s\}$.

Proposition 11 Given two k-CSTs $\mathbf{S}$ and $\mathbf{T}$ with $n \in N^{\mathbf{S}}, m \in N^{\mathbf{T}}$ and $s \in succ^{\mathbf{S}}(n)$, $t \in succ^{\mathbf{S}}(m)$ such that $n \curvearrowright_B m$ with $s \curvearrowright_F t$, then

$$cones(remove(\mathbf{S}, \langle n, s\rangle)) \subseteq cones(\mathsf{CST\_Minimization}(\mathbf{S}, \mathbf{T})).$$

Finally, simplified version of the rule of Definition 54 that requires only a local test for every node of a k-CST is given as follows.

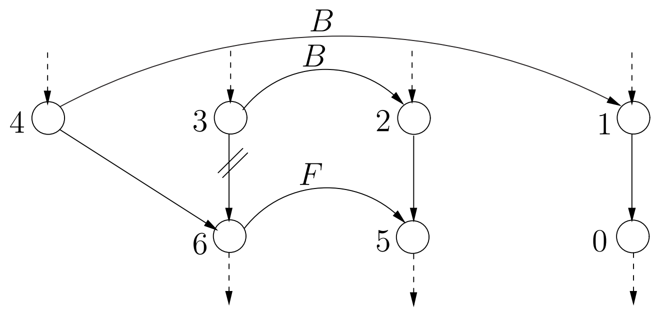

Figure 4.12: Removing edge $3 \rightarrow 6$ changes the backward simulation.

Definition 56 (Local Edge Removal) Given the k-CST $\mathbf{S}$, let us assume that the nodes $n, m, s, t \in N^{\mathbf{S}}$ with $s \in succ^{\mathbf{S}}(n)$ and $t \in succ^{\mathbf{S}}(m)$ $(s \neq t \vee n \neq m)$ are such that $val^{\mathbf{S}}(m) \sqsubseteq val^{\mathbf{S}}(n)$, $val^{\mathbf{S}}(t) \sqsubseteq val^{\mathbf{S}}(s)$,$succ^{\mathbf{S}}(s) \subseteq succ^{\mathbf{S}}(t)$ and $father^{\mathbf{S}}(n) \subseteq father^{\mathbf{S}}(m)$. Then, $local\_remove(\mathbf{S}, \langle n, s\rangle)$ is the pre-k-CST in which the successor relation $succ'$ is obtained from $\mathbf{S}$ by setting $succ'(n) = succ^{\mathbf{S}}(n) \setminus \{s\}$.

Similarly to the case of Definition 54, we have the following proposition.

Proposition 12 Given a k-CST $\mathbf{S}$ with the nodes $m, n \in N^{\mathbf{S}}$ and $s \in succ^{\mathbf{S}}(n)$, $t \in succ^{\mathbf{S}}(m)$ $(s \neq t \vee n \neq m)$ such that $val^{\mathbf{S}}(m) \sqsubseteq val^{\mathbf{S}}(n)$, $val^{\mathbf{S}}(t) \sqsubseteq val^{\mathbf{S}}(s)$,$succ^{\mathbf{S}}(s) \subseteq succ^{\mathbf{S}}(t)$ and $father^{\mathbf{S}}(n) \subseteq father^{\mathbf{S}}(m)$, then

$$cones(\mathbf{S}) = cones(local\_remove(\mathbf{S}, \langle n, s\rangle)).$$

Let us consider again Figure 4.10(a). If we apply the rule induced by Definition 56, the edge between n and s will be removed only if m has at least the same predecessors than n and if t has at least the same successors than s, which means that a path leading to a predecessor node of n also leads to a predecessor of m and the sub-path starting from the successors of s is also a sub-path starting from a successor of t. This is a more restrictive rule than the rule induced by Definition 54. Indeed, this last rule also allows us to removed the edge between n and s if paths leading to n are greater than paths leading to m and when sub-paths from the successors of s are greater than sub-paths starting from the successors of t. So, it is not necessary that m and t share, respectively, predecessor nodes with n and successor nodes with s.

Though less effective than the rule of Definition 54, the rule of Definition 56 can be paired with it in order to simplify the computation of the simulation relations. For instance, the rule induced by Definition 56 can be first applied to prune the k-CST with a low cost, and then the rule induced by Definition 54 is applied on the resulting (possibly smaller) pre-k-CST.

In the next chapter, we present how to use k-CST to verify MTNs that are abstractions of multi-threaded JAVA programs.

Chapter 5

Symbolic Verification Algorithms

We present in this chapter two algorithms for the verification of safety properties on MTNs. Safety properties can often be reduced to reachability properties. This means that safety properties (on a MTN) correspond to questions like

Is it possible to reach a set of markings U from a set S_0 of initial markings ?

Most of the time, the set U is a set of Bad markings corresponding to some bad states that should never occur in the system described by the MTN and $\mathbf{S}_0$ describes the set of initial configurations of the system.

In this thesis we focus on a special class of safety properties where the target set of markings is upward closed. These properties are called *upward closed properties*. The first approach to decide upward closed properties is to start from the initial markings and then compute a rough over-approximation of the set of reachable markings. This over-approximation is described through a *(weak) coverability set*. The second approach consists in computing from the target upward closed set U the set of markings from which U is reachable by firing a finite sequence of multi-transfers. Let us briefly describe the algorithms used in each of the two cases.

Backward Approach. Backward approach to verify the reachability of a set U from a set of initial states S_0 consists in starting from U and to iteratively compute the set of states from which U is reachable.

Figure 5.1 summarizes this approach. In this approach, an operation *pre* on states, that returns the predecessor states of the states given as parameters, is used.

First, *pre* is applied on U to compute the set of states from which U is reachable in at most one transition. Then, *pre* is applied on the resulting set of states to compute the set of states from which U is reachable by firing at most two transitions, etc. *pre* is applied until no new states are computed. More precisely, if U_i is the set of states from which U is reachable in at most i transitions, then we stop at the $(i+1)th$ step if $U_{i+1} \subseteq U_i$. In the general case, when this algorithm is applied on infinite state models, like MTN, this algorithm is not guaranteed to terminate.

Finally, if some initial states are in the set of states computed by applying *pre*, we conclude that U is reachable from S_0. Otherwise, we conclude that U is not reachable from S_0.

Notice that the test of inclusion can be done after each application of the *pre* operation. In this case, the algorithm stops when its detects that S_0 intersects the set computed by application of *pre* or when the fixpoint is reached. This allows the algorithm to terminate even in the case where the fixpoint does not converge, if U is reachable from S_0.

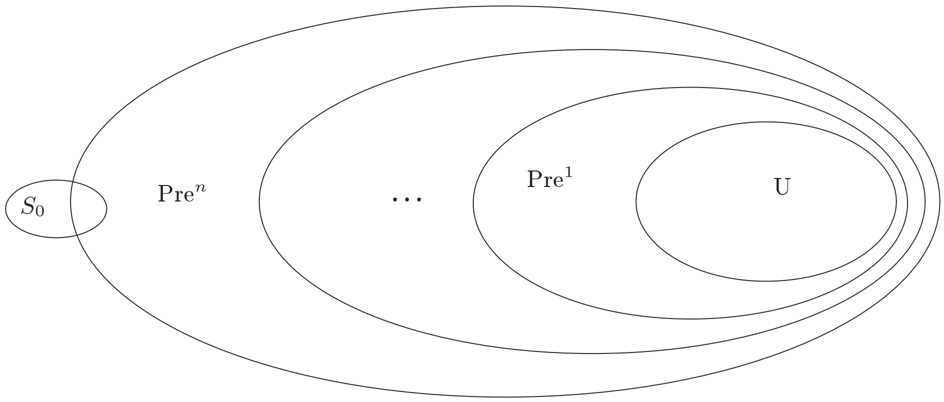

Figure 5.1: Backward Approach.

Forward Approach. Instead of starting from the set of states to be reached, forward approach to verify the reachability of a set U from a set of initial states S_0 consists in starting from S_0 and to iteratively compute the set of states reachable from S_0.

Figure 5.2 summarizes this approach. In this approach, an operation *post* on states, that

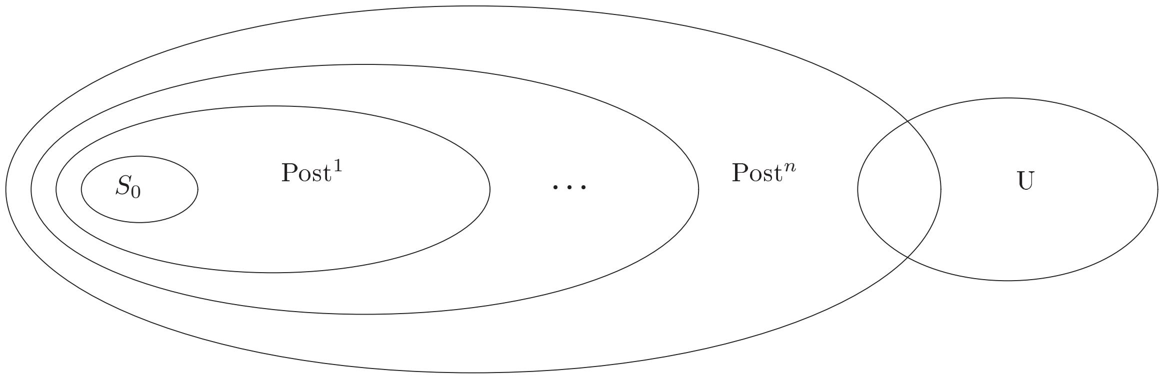

Figure 5.2: Forward Approach.

returns the successor states of the states given as parameters, is used.

Forward approach works in a similar manner than backward approach. First, *post* is applied on S_0 to compute the set of states reachable from S_0 in at most one transition. Then, *post* is applied on the resulting set of states to compute the set of states reachable from S_0 by firing at most two transitions, etc. *post* is applied until no new states are computed. More precisely, if S_i is the set of states reachable in at most i transitions from S_0, then we stop at the $(i+1)th$ step if $S_{i+1} \subseteq S_i$. As in the backward case, reaching the fixpoint is not guaranteed when applying this approach on infinite state models like MTN.

Finally, if some states in U are in the set of states computed by applying *pre*, we conclude that U is reachable from S_0. Otherwise, we conclude that U is not reachable from S_0. As in the backward case, we can test *on the fly*, i.e. during the computation of the fixpoint, if U has an empty intersection with the sets computed by application of *post*. In that case, the algorithm stops either if U has a non-empty intersection with the computed state space or if the fixpoint is reached.

In the rest of this chapter, we will focus on the backward and forward approaches applied on MTN when the set of Bad states is an upward closed set. As an introduction, we start with a brief comparison of the two approaches applied on Petri nets and their monotonic extensions, i.e. extensions that maintain the monotonicity property, like MTN, Petri nets with transfers [Cia94], Petri nets with reset [Bil91], etc. The concepts that are briefly presented here are more extensively presented in the rest of the chapter.

Conceptual Comparison We recall some facts presented in [KM69, Fin93, ACJT96, FS01] about the forward and backward approaches for the verification of coverability properties (the reachability of upward closed set of markings) of infinite states Petri nets and their monotonic extensions like MTN, Petri nets with transfers [Cia94], Petri nets with reset [Bil91], etc.

Forward	**Backward**
Over-approximation of successors	Exact set of predecessors
Not Robust	Robust
Downward closed Sets	Upward closed Sets
Acceleration	No Acceleration
Safety, Place boundedness	Safety
Depends on initial markings	Depends on bad markings

Table 5.1: Conceptual differences between forward and backward search.

Approximation of the computation. For Petri nets and their monotonic extensions like MTN, theoretical results [ACJT96, FS01] ensure the termination of the backward approach when the set of bad states is an upward closed set of markings. In that case, the backward approach computes the exact set of predecessors of an upward closed set of bad markings. On the other hand, the forward approach is not guaranteed to terminate. To ensure termination in practice, some techniques have been developed for Petri nets. Those techniques do not compute the exact set of reachable markings but a downward closed over-approximation of the set of reachable markings.

Robustness. Backward approach is robust for extensions of Petri nets that maintain monotonicity of the model in the sense that termination is guaranteed in theory for all the monotonic extensions of Petri nets. On the other hand, the techniques developed to ensure termination of the forward approach on Petri nets cannot always be extended to monotonic extensions of Petri nets, as we will see in the following. In consequence, we will only refer to Petri nets when talking about the forward approach in the other points.

Sets. Due to the monotonicity, starting from an upward closed set, the application of pre preserves the upward closure of the set. So, the backward search manipulates upward closed sets. In the forward case, as already mentioned in the previous points, techniques to ensure termination in theory have been developed for Petri nets. Those techniques manipulate generator sets of downward closed sets, and hence (slightly modified) forward approach for Petri nets manipulates downward closed sets.

Techniques to guarantee termination. Starting from an upward closed set and applying the pre operation, it is guaranteed to reach a fixpoint after a finite number of iterations, due to the monotonicity property of Petri net and the monotonic extensions. As already mentioned in the previous points, forward approach needs techniques to ensure termination in practice. Those techniques are based on an acceleration function (a function allowing us to compute an infinite set of markings in a finite number of steps) to reach a fixpoint when the net has an infinite state space.

Properties. Only coverability properties can be decided with the backward algorithm but the algorithms implementing the (slightly modified) forward approach allow us to solve in addition other problems like the *boundedness* and *place boundedness* problems.

Dependence on the search. Forward approach starts from the initial configurations that we want to verify. In consequence, the state space explored by the forward approach is the same for any properties (except if we change the initial configurations of the system). On the other hand, with the backward approach, the computation depends on the property to be verified. In consequence, the state space explored by the backward approach changes when considering different upward closed sets.

In the next section, we present symbolic model checking and then, in the following sections, we present a backward and a forward symbolic approach that uses k-ucCST and k-dcCST for the verification of upward closed properties on MTN.

The content of this chapter is based on the content of the papers [DRB01, DRVB02, FRSVB02].

5.1 Symbolic Model Checking

Model checking is a verification technique that consists in analysing models by making an exhaustive exploration of their state space.

Symbolic model checking has been first applied on finite state systems to avoid the *state space explosion* observed when applying verification algorithms. The state space explosion problem consists in having a huge state space, even for simple systems. Instead of handling reachable states one by one as in the enumerative case, the symbolic approach consists in manipulating sets of states with operations on data structures representing them. The symbolic approach using BDD has been applied successfully to the verification of large finite state systems [BCM$^+$90]; this is due to the fact that binary decisions diagrams are efficiently encoded by merging isomorphic subtrees and removing useless nodes [Bry86].

A few years ago, the symbolic approach has been then applied on the verification of infinite state systems. In this case, infinite sets of states must be described by adequate data structures. IDDs [ST99] and DDDs [MLAH99] are extensions of BDDs where nodes and/or edges determine more complex values than boolean values. In IDDs edges are labeled by intervals on integers and nodes in DDDs are labeled by constraints on variables of the form $x - y \geq c$ instead of variables. These two data structures allow to represent some restricted classes of infinite sets in $\mathbb{Z}^n$ or $\mathbb{R}^n$. A more complex class of sets, i.e. sets that can be described with Presburger formula, can be encoded and manipulated with automata-based data structures like NDDs [Boi99] or convex Polyhedra represented either by a *system of linear inequality* or by a *system of generators*, i.e. two finite sets V and R of tuples in $\mathbb{N}^k$, $\mathbb{Z}^k$, or $\mathbb{Q}^k$ such that the polyhedra is the set $\{v|v = \sum_{s_i \in V} c_i \cdot s_i + \sum_{r_j \in R} d_j \cdot r_j$ with $c_i \in \mathbb{N}, d_j \in \mathbb{N}$ and $\sum_i c_i = 1\}$.

Automata-based data structures are also provided for the verification of pushdown automata [FWW97], FIFO channels systems [BG96] and parameterized systems[ABJN99, JN00].

For timed systems, special Matrix representations, called DBM (Difference Bound Matrices), have been developed. Each line and each column corresponds to a clock (a natural variable that growths linearly with time). Each element at the ith line and jth column is a tuple $\langle c, <\rangle$ or $\langle c, \leq\rangle$ where $c \in \mathbb{N}$ and is interpreted as the constraints $x_i - x_j \bullet c$ ($\bullet \in \{<, \leq\}$). Actually, DDDs are an efficient way to represent sets of DBMs. DBMs have also been extended with parameters and constraints on these parameters in [AAB00] to describe sets of states of parameterized timed automata extended with counters.

These data structures have been implemented and tested on practical examples in tools like HyTech [HHWT97], TReX [ABS01], Brain [RV02] and Fast [BFLP03].

In the following, we define symbolic algorithms for the verification of upward closed properties on MTN. More precisely, we explain in the next sections how to use k-ucCSTs and k-dcCSTs to verify upward closed set properties on MTNs.

5.2 Backward Approach

Backward Approach for monotonic systems has been studied in [ACJT96, FS01]. Given a system with a finite set of initial markings S_0, a well-quasi ordering $\sqsubseteq$ on its states such that the underlying transition relation of the system is monotonic with respect to $\sqsubseteq$, it has been proved in [ACJT96, FS01] that the reachability of an upward closed set U according to $\sqsubseteq$ is decidable. The generic algorithm 6 is proposed to solve this problem.

This algorithm is based on the following properties:

(i) all the upward closed sets of states of the system have a finite number of minimal elements. This property is guaranteed as the partial order $\sqsubseteq$ on states is a well-quasi ordering. Intuitively, from each state s of the system it not possible to construct an infinite sequence of states $s_0 s_1 \ldots$ with $s_0 = s$ and $s_i \sqsupset s_{i+1}$ for all $i \geq 0$. As a consequence, each state is covered by at least one minimal element and hence the set of minimal elements of the upward closed set is a generator set of this set. As this set only contains incomparable states with respect to $\sqsubseteq$, then it is guaranteed to be finite (otherwise, as $\sqsubseteq$ is a well quasi ordering, it should contain two states s_i and s_j with $s_i \sqsubseteq s_j$, which is not possible);

(ii) the set of predecessors of an upward closed set is guaranteed to be also upward closed. This property is guaranteed by the monotonicity of the system.

Algorithm 6 computes the predecessor set of states applying the predicate pre_{min}. Given a state s, $\mathsf{pre}_{min}(s)$ returns the set of minimal elements of the predecessor set of the set $\{s'|s \sqsubseteq s'\}$. In the same way, $X \cup_{min} Y$ returns the minimal elements of the union of the upward closed sets defined by X and Y.

The fixpoint is reached when the upward closure of F is included into the upward closure of $fixpoint$, which holds when all the minimal elements of F are greater than at least one minimal element of $fixpoint$.

Finally, to decide if the upward closed set U is reachable from S_0, it suffices to find a minimal state in $fixpoint$ lesser than an element in S_0.

The following theorem states that Algorithm 6 can be used to decide the reachability of an upward closed set.

Theorem 25 (From [ACJT96, FS01]) Let $\mathcal{SYS}$ be a monotonic system with the set of states S and $\sqsubseteq$ be a well quasi ordering on the states such that $\mathcal{SYS}$ is monotonic with respect to $\sqsubseteq$, S_0 be a finite set of initial states and let $U \subseteq S$ be an upward closed set of states according to $\sqsubseteq$.

Then, Algorithm 6 applied on $\mathcal{SYS}$, S_0 and U terminates and answers *yes* if U is reachable from S_0 in $\mathcal{SYS}$, *no* otherwise.

Algorithm 6: Upward_Closed_Backward_Algorithm

Input: $\mathcal{SYS}$: a monotonic system; U : the set of minimal states of an upward closed set of states of $\mathcal{SYS}$; S_0 : set of initial states of $\mathcal{SYS}$
Result: *boolean*
begin
  $fixpoint = \emptyset$;
  $F = U$;
  while $\exists s \in F : \forall s' \in fixpoint : s' \not\sqsubseteq s$ **do**
    $fixpoint = fixpoint \cup_{min} F$;
    $oldF = F, F = \emptyset$;
    foreach $s \in oldF$ **do**
      $F \cup_{min} \mathsf{pre}_{min}(s)$;
  if $\exists s \in S_0, s' \in fixpoint : s' \sqsubseteq s$ **then**
    return *true*;
  else
    return *false*;
end

In the rest of this chapter, we will always consider that the set of initial states S_0 is a downward closed set of markings described through its finite set of limit elements.

For instance, if we want to decide the reachability of U from one initial marking $\mathbf{m}_0$ for a Petri net N, then we will consider that $S_0 = dcones(\mathbf{m}_0)$. Actually, that can be done because the monotonicity property of Petri nets (and of some of their extensions, like MTN) ensures that we have $\mathsf{Reach}_N(\mathbf{m}_0) \cap U = \emptyset$ if and only if $\mathsf{Reach}_N(dcones(\mathbf{m}_0)) \cap U = \emptyset$. This is particularly interesting when we consider *parameterized initial markings*.

Figure 5.3 shows a Petri net modeling a mutual exclusion protocol. The place p_1 describes a waiting state of processes, the places p_4 and p_5 describe critical sections of processes and the places p_2 and p_3 correspond to boolean variables used to ensure the mutual exclusion of the critical sections. As the protocol is designed for any number of processes, the initial marking $\langle K, 1, 1, 0, 0\rangle$ contains a parameter K meaning that there is some (at least one) processes in the waiting state in the initial state. Parameterized markings describe an infinite set of markings. However, the downward closure of these sets will be finitely represented by limit elements. For

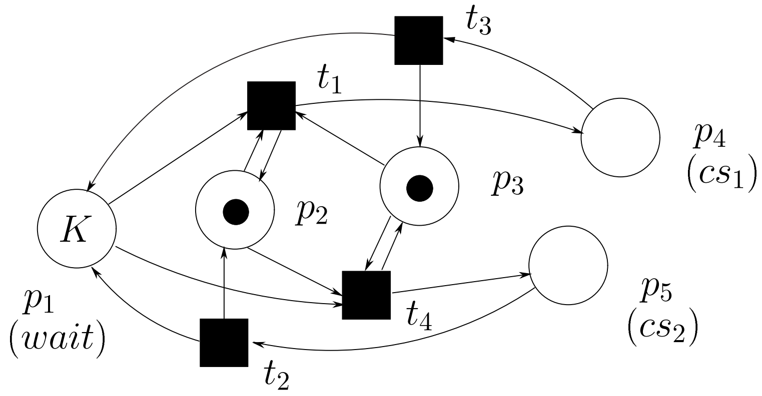

Figure 5.3: An example of Petri Net with parametric initial marking $\langle K, 1, 1, 0, 0\rangle$, $K \geq 1$.

our example, the verification algorithms will be applied on a downward closed set of initial markings described by the limit element $\langle +\infty, 1, 1, 0, 0\rangle$.

Notice that, in our case, parameters will be particularly interesting to describe families of global machines describing the behaviour of Java programs (see Chapter 2 and Chapter 3).

To make the generic algorithm 6 effective [1], sub-algorithms for performing the operations and tests on the set of minimal elements needed by the algorithm must be defined. This has already been done for several kinds of infinite state systems like Lossy FIFO channels [AJ93], Petri nets [DR00, AIN00], Broadcast protocols [EFM99], etc.

We present in this section backward algorithms to verify upward closed properties on Petri nets and then extend them to MTNs. We then show how to apply symbolic techniques on theses algorithms. More precisely, we will use k-ucCSTs to encode minimal elements of upward closed sets manipulated by algorithms. This means that we have two levels of symbolic encoding. The first level consists in manipulating upward closed sets through their set of minimal elements and the second level consists in manipulating this set of minimal elements through k-ucCSTs.

5.2.1 Symbolic Backward Reachability Algorithm for Petri Nets

Following, e.g. [Rei86, STC98], a Petri Net can be represented via a *vector addition system* (VAS for short), i.e., a machine model that consists of n variables $x_1, \ldots, x_n$ (one for each place) ranging over non-negative integers, and m *guarded commands* (one for each transition [2]) defined over the data variables. The guarded command associated with the *ith* transition contains a guard $x_j \geq c_{ij}$ and an assignment $x_j := x_j + d_{ij}$ for every place p_j ($c_{ij} \in \mathbb{N}, d_{ij} \in \mathbb{Z}$). The guard is such that

- $c_{ij} \geq 0$ if $d_{ij} \geq 0$: when reasoning backward, it avoids to assign negative values to x_j;
- $c_{ij} \geq -d_{ij}$ if $d_{ij} < 0$: when reasoning forward, it avoids to assign negative values to x_j.

Markings then represent the current state of the n-counters associated with the places. Having this view in mind, it should be very easy to see how to apply $\mathcal{U}$-formulas for representing *covering* problems of Petri nets, and thus how to use k-ucCSTs to efficiently solve them.

[1] Effective means that we can build a program from this Algorithm.
[2] When we only consider Petri nets, multi-transfers are called transitions.

Given a Petri Net N and an upward-closed set of markings U, we will show that we can define a $\mathcal{U}$-formula to represent the set of *predecessors* of markings in U with respect to the transitions in N. Actually, we already described in Section 4.2 how to represent an upward-closed set U as a $\mathcal{U}$-formula φ_U. It remains us to show how to construct the $\mathcal{U}$-formula that represents $\mathsf{pre}(U)$.

Let us define the formula $\varphi[x_i + c/x_i]$ with $c \in \mathbb{Z}$ built from φ by replacing all the occurrences of x_i by $x_i + c$ and then simplifying the resulting formula. For instance, the formula $(x_1 \geq 3 \wedge x_2 \geq 1)[x_1 + 2/x_1]$ is equivalent to the formula $(x_1 \geq 1 \wedge x_2 \geq 1)$. Intuitively, this corresponds to remove c from the constants associated to the constraints on x_i.

Definition 57 (The $\mathcal{U}$-formula for $\mathsf{pre}(U)$) Let $\varphi_i = \wedge_{j=1\dots n} x_j \geq c_{ij}$ be the guard of the i-th guarded command (transition) and let $x_j := x_j + d_{ij}$ be the action associated with the i-th guarded command (transition) and to the jth variable (place). The following $\mathcal{U}$-formula can be used to represent $\mathsf{pre}(U)$:

$$\Phi_{\mathsf{pre}(U)} = \bigvee_{i=1}^{m} (\varphi_i \wedge \Psi_i)$$

where $\Psi_i = \Phi_U[x_1 + d_{i1}/x_1] \dots [x_n + d_{in}/x_n]$

It is easy to verify that $[\![\Phi_{\mathsf{pre}(U)}]\!] = \mathsf{pre}(U)$. Notice that, according to results of section 4.2, if Φ_U is in disjunctive form we can use the specific algorithms on $\mathcal{U}$-formula in disjunctive form to obtain a formula (equivalent to) $\Phi_{\mathsf{pre}(U)}$ that is still in disjunctive form. In the rest of this section, we always use $\mathcal{U}$-formulas in disjunctive form. Based on this idea, in order to verify problems that can be expressed via upward-closed sets, we can build the symbolic backward reachability algorithm BS (Algorithm 7). The algorithm works using $\mathcal{U}$-formulas and their operations. The initial downward closed set of configurations is represented by an extended valuation (the extension to the general case where the downward closed set is represented by several extended valuations is obtained by changing the last test of the algorithm, i.e. by testing if one of the extended valuations satisfies Φ).

By construction, the intermediate results are always in disjunctive form. At each step of the algorithm, we use subsumption as termination test.

Notice that the satisfiability test on the extended valuation $\mathbf{t}_I$ and the $\mathcal{U}$-formula Φ corresponds to test if the intersection between the denotation of Φ and the downward closed set generated by $\mathbf{t}_I$ is empty.

Indeed, since Φ is in disjunctive form, i.e. $\Phi \equiv x_{11} \geq c_{11} \wedge \dots \wedge x_{1k} \geq c_{1k} \vee \dots \vee x_{n1} \geq c_{n1} \wedge \dots \wedge x_{nk} \geq c_{nk}$, we have $\mathbf{t}_I \models \Phi$ if and only if $\exists i.1 \leq i \leq n$ and $\langle c_{i1}, \dots, c_{ik} \rangle \preccurlyeq \mathbf{t}_I$. If we have $\langle c_{i1}, \dots, c_{ik} \rangle \preccurlyeq \mathbf{t}_I$ for some i such that $1 \leq i \leq n$, then we can construct a marking $\mathbf{t} \in \mathbb{N}^k$ from $\mathbf{t}_I$ such that $\mathbf{t} \preccurlyeq \mathbf{t}_I$ and $\mathbf{t} \models \Phi$ (it suffices to replace in $\mathbf{t}_I$ all the $+\infty$'s by $max(c_{i1}, \dots, c_{ik})$). Moreover, if we have $\langle c_{i1}, \dots, c_{ik} \rangle \not\preccurlyeq \mathbf{t}_I$ for all i such that $1 \leq i \leq n$, then for all $\mathbf{t} \preccurlyeq \mathbf{t}_I$ we have $\langle c_{i1}, \dots, c_{ik} \rangle \not\preccurlyeq \mathbf{t}_I$ for $1 \leq i \leq n$.

Algorithm 7 can be extended to the case, where we have several initial extended valuations. In that case, the algorithm returns *true* if one of the extended valuations satisfies Φ, otherwise it returns *false*.

Interestingly, this algorithm coincides with the backward search scheme implemented by the symbolic model checkers HyTech [HHWT97] and ALV [BGP97]. These tools use *linear*

Algorithm 7: BS

Input: $\mathbf{t}_I \in (\mathbb{N} \cup \{+\infty\})^k$, $\Phi_U : \mathcal{U}-formulas$
Result: *boolean*
begin
  $\Phi \leftarrow \Phi_U$;
  $\Psi \leftarrow empty$ $(i.e.\Psi = x_1 \geq +\infty \wedge \ldots \wedge x_n \geq +\infty)$;
  while $\Phi \not\models \Psi$ **do**
    $\Psi \leftarrow \Phi$;
    $\Phi \leftarrow \Psi \vee \mathsf{pre}(\Psi)$ $(\mathsf{pre}(\Psi)\ as\ in\ Def.57)$;
  if $\mathbf{t}_I \models \Phi$ **then**
    return *true*;
  else
    return *false*;
end

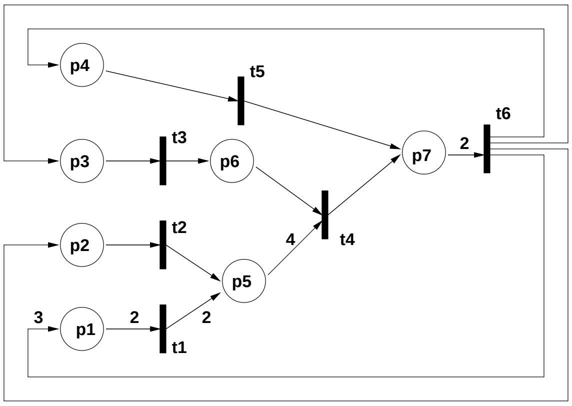

Figure 5.4: The table manufacturing system.

arithmetic constraints as the assertional languages to represent *infinite* collections of system states (i.e. assignments of reals or integers to a given number of data variables). Furthermore, they internally maintain formulas in disjunctive form (linear arithmetic constraints are defined as conjunctions of atomic formulas). Thus, they can be used as effective implementation platforms for the backward search algorithm based on $\mathcal{U}$-formulas described above. In [DEP99], this technology is tested on some Petri Net examples of small size. We report on these experiments in the following section.

Symbolic State Explosion

The *table manufacturing system* of Teruel [Ter94] (see Figure 5.4) is a production system modeled via a Petri Net with 7 places and 6 transitions. The number of tokens added to or removed from each place by the transitions vary from 2 to 4. This Petri Net can be modeled using a VAS with 13 variables, 7 for the places and 6 extra variables, say $T_1, \ldots, T_6$, to keep track of the number of firings of each transition. As shown in [Ter94], all deadlock-free initial markings are exactly the predecessors of the set of states where the first transition is fired at least 3 times, and all the others are fired at least twice (i.e. $U = T_1 \geq 3 \ \wedge \ \bigwedge_{j>1} T_j \geq 2$). In [DEP99], different general purpose constraint-based checkers have been used to compute the set of predecessors via backward reachability. As shown in Table 5.2, the comparison was carried out using HyTech, the tool based on the Omega library of Bultan (called here Omega), Gerber and Pugh [BGP97], and a specialized implementation proposed in [DEP99] and called here BitVector, in which collections of linear constraints are stored as a list of lists of bit-vectors. As shown in Table 5.2 (see columns BitVector, HyTech and Omega), the general purpose model checkers suffer from the symbolic state explosion, due to the way in which constraints are stored, i.e., maintaining them in disjunctive normal form.

On the contrary, the specialized data structures proposed in [DEP99] perform reasonably well. The data structure of [DEP99] is based on the following idea. A constraint like $x \geq c$ is represented via a pair $\langle b, c\rangle$ in which b is a bit vector. Variable x_i is represented as the unit bit vector with 1 in position i (for n variables, we need then n bits). A conjunction of constraints can be stored then as a *list of such pairs*. Finally, a collection of constraint is stored as a *list of list of pairs*. As an example, the $\mathcal{U}$-formula in disjunctive normal form over the three variables $\langle x_1, x_2, x_3\rangle$ $\varphi_1 \vee \varphi_2$, where $\varphi_1 = (x_1 \geq 2 \wedge x_2 \geq 1 \wedge x_3 \geq 4)$ and $\varphi_2 = (x_1 \geq 2 \wedge x_2 \geq 4 \wedge x_3 \geq 1)$ is represented as follows.

φ_1 :	$\langle 100, 2\rangle$	$\langle 010, 1\rangle$	$\langle 001, 4\rangle$
φ_2 :	$\langle 100, 2\rangle$	$\langle 010, 4\rangle$	$\langle 100, 1\rangle$

As shown in [DEP99], bit-a-bit operations can be used to handle the resulting representation. Although when applied to $\mathcal{U}$-formulas, the use of bit-vectors to represent variables might seem a waste of space, this representation can be applied to capture more sophisticated classes of constraints (see [DEP99]). As an example, we can represent constraints like $x_1 + x_2 + x_3 \geq 5$ via the pair $\langle 111, 5\rangle$.

Unfortunately, this symbolic representation of $\mathcal{U}$-formulas suffers from another potential source of redundancy. Note, in fact, that the $\mathcal{U}$-formula of our example can be written more compactly as $(x_1 \geq 2) \wedge ((x_2 \geq 1 \wedge x_3 \geq 4) \vee (x_2 \geq 4 \wedge x_3 \geq 1))$. Intuitively, what is missing in the data structure of [DEP99] is the possibility of *sharing* information among different disjuncts. This justifies the use of k-ucCSTs as a heuristics to obtain compact data structures for this kind of verification problems (see Table 5.3 for the obtained results and below for the explanations).

Example	Nb.trans.	U=Unsafe States	Steps	BitVector[1]	HyTech[1]	Omega[2]
Man. Sys. [Ter94]	6	$T_1 \geq 3$, $\wedge_{i>1} T_i \geq 2$	24	18m10s	1h55m25s	19h50m

[1] On a Sun Sparc 5.6. [2] On a Sun Ultra Sparc.

Table 5.2: Comparisons of Constraint-based verification tools.

k-ucCST-based Symbolic Backward Reachability

The first step toward a k-ucCST-based symbolic backward search algorithm consists in formulating the $\mathcal{U}$-based algorithm using the corresponding k-ucCST-operations. After this preliminary step, we will try to exploit the structure of k-ucCSTs in order to find optimizations that we can use during backward search. As mentioned in the previous Chapter (Section 4.5), it is often convenient to work on generic k-ucCSTs (i.e. in which we allow redundancies). As an example, in this case disjunction can be done in polynomial time. On the other hand, too many redundancies might cause an explosion of the space needed to store intermediate results. The trade off between the complexity of the handling and the compactness of the k-ucCSTs is a crucial aspect to be taken into account, e.g., in the computation of the k-ucCST-based symbolic predecessor operator. Another important point to consider is the complexity of the subsumption test used as the termination test for the fixpoint computation of pre^*. In the next sections we will discuss these two points and then present the resulting k-ucCST-based backward search algorithm. Let us start discussing the symbolic computation of pre.

A k-ucCST-based pre operation for Petri Nets

Given a k-ucCST $\mathbf{S}_\varphi$ representing a $\mathcal{U}$-formula in disjunctive form φ (hence an upward-closed set of markings), we would like to symbolically compute $\mathsf{pre}(\varphi)$ using operations on k-ucCSTs. In the following we will focus our attention on the backward effect of a single Petri Net transition M, represented via a guarded command in which the guard $\varphi_j = x_j \geq c_j$ and the action $x_j := x_j + d_j$ are associated with the j-th place.

Let $\mathbf{S}_\varphi$ be a k-ucCST, then the operator $\mathsf{k\text{-}ucCST\_Pre}(M, \mathbf{S})$ should satisfy the following property

$$cones(\mathsf{k\text{-}ucCST\_Pre}(M, \mathbf{S}_\varphi)) = cones(\mathbf{S}_\psi),$$

where $\psi = \varphi_M \wedge (\varphi[x_1 + d_1/x_1] \ldots [x_k + d_k/x_k])$, and $\varphi_M = \varphi_1 \wedge \ldots \wedge \varphi_n$ is the guard of the transition M. A possible implementation of this operation makes use of k-ucCSTs symbolic conjunction as follows:

$$\mathbf{S}_\psi = \mathsf{k\text{-}ucCST\_Intersection}(\mathbf{S}_{\varphi_M}, \mathbf{S}_{\varphi[x_1+d_1/x_1]\ldots[x_k+d_k/x_k]}).$$

The construction of the k-ucCST $\mathbf{S}_{\varphi[x_i+d/x_i]}$ from the k-ucCST $\mathbf{S}_\varphi$ consists in removing d from the value of nodes in the layer corresponding to x_i [3]. So the computation of the k-ucCST $\mathbf{S}_{\varphi[x_1+d_1/x_1]\ldots[x_k+d_k/x_k]}$ from $\mathbf{S}_\varphi$ consists in applying this treatment for x_1 with $d_1, \ldots, x_k$ with d_k.

The k-ucCST $\mathbf{S}_{\varphi[x_1+d_1/x_1]\ldots[x_k+d_k/x_k]}$ represents in a compact way the disjunction of constraints obtained after the backward application of the action of M. The final result is obtained by intersecting the resulting k-ucCST with the k-ucCST associated with the guard of M. Although the k-ucCST $\mathbf{S}_{\varphi_M}$ is a very special k-ucCST, i.e. it contains only one path, intersection makes this operation exponential in the size of $\mathbf{S}_\varphi$. Actually, this is also the cost of the search problem of symbolically computing $\mathsf{k\text{-}ucCST\_Pre}$ on k-ucCSTs. We will formally prove this claim in the following theorem.

[3] This could lead to nodes with negative values. Indeed, suppose in our example that $d = 2$ and there is a node in the layer corresponding to x_i with the value 1. After the operation, this node will have the value -1. In consequence, we consider for this explanation that the domain of values of nodes has been extended to integer values. Anyway, after the intersection with $\mathbf{S}_{\varphi_M}$ those negative values will disappear.

Theorem 26 Let $\mathbf{S}$ be a (irredundant or not) k-ucCST and M be a Petri Net transition. Then (unless P= NP), there is no polynomial algorithm to compute a (irredundant or not) k-ucCST $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathsf{k\text{-}ucCST\_Pre}(M, \mathbf{S}))$.

Proof. Let us first consider the case in which the input k-ucCST $\mathbf{S}$ is irredundant and the result is required to be irredundant, and then extend the proof to generic k-ucCSTs.

The case $\mathsf{k\text{-}ucCST\_Pre}(IRR) = IRR$**.** As in Section 4.5, we exhibit a polynomial *Turing* reduction from *validity*. Specifically, we build an OTM (Oracle Turing Machine) that uses k-ucCST_Pre as an oracle to decide *validity* in polynomial time.

1. Let $F = D_1 \vee \ldots \vee D_n$ be a propositional formula in DNF defined over k symbols. We first construct the k-ucCST $\mathbf{T}_{val(D_i)}$ for $i : 1, \ldots, n$ and $\mathbf{T}_{val(true)}$, i.e. the k-ucCSTs without the extra layers used in the proof of Theorem 15. This step can be done in *polynomial* time.

2. As a second step, we add two extra layers on the top of every newly computed k-ucCST $\mathbf{T}_{val(D_i)}$; let us call $\mathbf{T}_i$ the so obtained k-ucCST. The labels in the two extra layers are such that all resulting $\mathbf{T}_i$'s have generators that are not comparable to each others. Specifically, we simply add the prefix with labels $\langle i, n-i+1\rangle$ to every tuple in $\mathbf{T}_i$. Note that for each $\mathbf{T}_i$ we need only two nodes and two edges distributed in the two extra layers: the node in the first layer has label i, the node in the second layer has label $n-i+1$; and all paths going out from the root are forced to pass through the two nodes.

3. Now, we build the k-ucCST $\mathbf{T}_F$ by first merging all the $\mathbf{T}_i$'s in a single pre-sharing tree, and then reducing it in a k-ucCST by sharing all their *suffixes* using the polynomial time algorithm proposed in [Zam97] (see Appendix .1, Algorithm 27). From now on, we will interpret the resulting k-ucCST $\mathbf{T}_F$ as representing markings for the variables $x, y, x_1, \ldots, x_k$, where x, y are the variables associated with the two newly introduced extra layers, and $x_1, \ldots, x_k$ are the variables associated with the proposition symbols in F. Note that n is the largest value occurring in the constraints associated with x, y in $\mathbf{T}_F$.

4. We now define a Petri Net transition M having the following guard:
$$x \geq n \ \wedge y \geq n \ \wedge x_1 \geq 0 \ \wedge \ \ldots \ \wedge x_k \geq 0.$$
and the following action
$$x := x \ \wedge \ y := y \ \wedge x_1 := x_1 \wedge \ldots \wedge x_k := x_k.$$
This transition first removes and then puts back n tokens from x and y. Thus at least n tokens must be in x and y in order for M to be fired. The transition has no effects on the other variables.

5. Notice that the result of computing k-ucCST_Pre over the constraints associated with x and y in $\mathbf{T}_F$ is always the constraint $x \geq n \wedge y \geq n$ (by construction, the constants in $\mathbf{T}_F$ associated with x and y range in $1, \ldots, n$). As a consequence, if we apply the oracle to M and $\mathbf{T}_F$, we will obtain an irredundant k-ucCST in which: (1) there is only one path associated with the variables x and y with labels $\langle n, n\rangle$, as M removes

n tokens from those two places this is the minimal number of tokens we need to fire it; (2) every path corresponding to a valuation for the propositional symbols of F occurs at most once. Point (2) is a consequence of (1) and of the fact that the oracle returns an irredundant k-ucCST.

6. We build now the k-ucCST $\mathbf{T}_{val(true)}$ associated with all possible evaluations of the propositional symbols in F, and then we add the prefix $\langle n, n\rangle$ to the tuples in $\mathbf{T}_{val(true)}$ as for the $\mathbf{T}_i$'s. Let $\mathbf{S}_{true}$ be the resulting k-ucCST. By construction, we obtain then that F is *valid* if and only if $cones(\textsf{k-ucCST\_Pre}(M, \mathbf{T}_F)) = cones(\mathbf{S}_{true})$. Finally, we note that, since both k-ucCSTs are irredundant, we can compute their equivalence in polynomial time (see Theorem 4.2).

The resulting Turing reduction is polynomial, thus coNP hardness for our search problem follows from the coNP completeness of *validity*.

The case k-ucCST_Pre$(GEN) = GEN$**.** We can extend the proof of coNP hardness when k-ucCST_Pre works on *generic* k-ucCSTs as follows. Instead of labeling the extra layers of the k-ucCST $\mathbf{T}_i$ using the prefix $\langle i, n-i+1\rangle$, we always use the prefix $\langle i, i\rangle$. As a consequence, we will get a k-ucCST $\mathbf{T}_F$ that may contain redundant tuples. If we now apply the oracle (working on any k-ucCST) we will obtain a k-ucCST k-ucCST_Pre$(M, \mathbf{T}_F)$ that may also contain redundant tuples. Now, we build $\mathbf{S}_{true}$ exactly as in the case k-ucCST_Pre$(IRR) = IRR$. By construction, however, we notice that F is *valid* if and only if $cones(\textsf{k-ucCST\_Pre}(M, \mathbf{T}_F))$ contains all generators of $cones(\mathbf{S}_{true})$. Indeed, if F is valid, then we have $cones(\mathbf{S}_{true}) =$ k-ucCST_Pre$(M, \mathbf{T}_F)$. This means that if $cones(\textsf{k-ucCST\_Pre}(M, \mathbf{T}_F))$ contains more paths than $cones(\mathbf{S}_{true})$, those paths correspond to redundancies. Following Figure 4.1, we can check this property by using containment of the flat denotations, i.e. by testing $elem(\mathbf{S}_{true}) \subseteq elem(\textsf{k-ucCST\_Pre}(M, \mathbf{T}_F))$ in polynomial time. The resulting Turing reduction is again polynomial. Thus, coNP hardness for our search problem follows from the coNP completeness of *validity*. □

As shown in the proof of the previous theorem, the problems with computing k-ucCST_Pre arises from the presence of guards in a Petri Net transition. Nevertheless, we will show later that in practice computing k-ucCST_Pre is never a bottleneck for the k-ucCST-based search engine. Intuitively, the reason is that in practice only few labels collapse to the same value. In most cases, the application of k-ucCST_Pre amounts to a mere substitution operation.

k-ucCST-driven Optimizations of k-ucCST_Pre

Following the definition of k-ucCST_Pre, we directly get the following symbolic algorithm to compute the pre-image of a set of markings:

$$\textsf{k-ucCST\_Pre}(\mathbf{S}) = \cup_{M \in \mathcal{B}}\ \textsf{k-ucCST\_Pre}(M, \mathbf{S})$$

where $\mathcal{B}$ is the set of transitions of our Petri net.

In this section, we will focus on some optimizations of this algorithm. In the following, we will extend the mapping $cones(\cdot)$ to denote the rich denotation of a pre-k-ucCST (i.e. in which condition 5 or 6 may not hold). First, we can notice that it is easy to characterize the set of predecessors of an upward-closed set with one minimal marking, as described by the following proposition.

Proposition 13 Let $\mathbf{m}$ be a marking in $\mathbb{N}^k$ and M be a transition of Petri nets. Then, we have $\mathsf{pre}(M, cone(\mathbf{m})) = cone(\mathbf{m}')$ where $\mathbf{m}'(i) = max(c_i, \mathbf{m}(i) - d_i)$ for all $1 \leq i \leq k$.

Following Proposition 13, we can compute the minimal marking of the predecessors set for a transition M via independent computations on each component. As the denotation of a k-ucCST is the union of the upward-closed set described by its paths, we can define an algorithm based on this idea. More precisely, given a k-ucCST $\mathbf{S}$, we compute a pre-k-ucCST that has the same rich denotation as k-ucCST_Pre$(M, \mathbf{S})$: we set the value of a node n in the i-th layer to the following value:

$$val'(n) = max(c_i, val(n) - d_i)$$

The max corresponds to the conjunction between the guard and the result of the backward application of the action in M to x_i. We use afterwards the algorithms given in [Zam97] (see Appendix .1) to adjust the two conditions 5 and 6. Although the algorithm for ensuring condition 5 is exponential in the worst case, this method performs well in practice.

Algorithm 8: Symbolic$_{\mathsf{BS}}$

```
Input: ⟨P, B⟩ : Petri net, S_U : k-ucCST, m_0 ∈ (ℕ ∪ {+∞})^k
Result: boolean
begin
    S ← S_U;
    T ← k-ucCST_Empty_set;
    while not(k-ucCST_Subsumption(S, T)) do
        T ← k-ucCST_Union(T, S);
        R ← k-ucCST_Empty_set;
        foreach M ∈ B do
            N ← k-ucCST_Pre(M, S);
            R ← k-ucCST_Union(N, R);
        S ← R;
    return k-ucCST_Sat(m_0, T);
end
```

Algorithm 8 is the backward reachability algorithm based on k-ucCSTs. The operations k-ucCST_Subsumption, k-ucCST_Union and k-ucCST_Sat correspond respectively to the operations CST_Subsumption, CST_Union and CST_Sat (extended to limit elements in $(\mathbb{N} \cup \{+\infty\})^k$) applied on k-ucCSTs. The algorithm computes the k-ucCST_Pre operation on the variable $\mathbf{S}$ associated with the k-ucCST describing the set of minimal markings computed at every iteration until the fixpoint is reached (k-ucCST_Subsumption$(\mathbf{S}, \mathbf{T})$ is evaluated to true). When the fixpoint is reached, the algorithm tests if $\mathbf{m}_0$ satisfies $\mathbf{T}$. Notice that $\mathbf{m}_0$ can contain some $+\infty$ and corresponds to what we call an extended valuation at the beginning of Section 4.2. In this case, the fact that $\mathbf{m}_0$ satisfies $\mathbf{T}$ means that there is a marking in the downward closure of $\mathbf{m}_0$ that satisfies $\mathbf{T}$.

The k-ucCST stored in $\mathbf{S}$ may contain several redundant paths. The following proposition can help us to reduce its size.

Proposition 14 Let $\mathbf{m}$ and $\mathbf{m}'$ be two markings, and let M be a transition of a Petri net. If $\mathbf{m} \preccurlyeq \mathbf{m}'$ holds, then we have

i) $\mathsf{k\text{-}ucCST\_Pre}(M, cone(\mathbf{m}')) \subseteq \mathsf{k\text{-}ucCST\_Pre}(M, cone(\mathbf{m}))$;

ii) $\mathsf{k\text{-}ucCST\_Pre}^*(cone(\mathbf{m}')) \subseteq \mathsf{k\text{-}ucCST\_Pre}^*(cone(\mathbf{m}))$.

where $\mathsf{k\text{-}ucCST\_Pre}^*(cone(\mathbf{m}')) = \bigcup_{i\geq 0} \mathsf{k\text{-}ucCST\_Pre}^i(cone(\mathbf{m}'))$ and $\mathsf{k\text{-}ucCST\_Pre}^i(cone(\mathbf{m}'))$ is the set obtained after i successive applications of $\mathsf{k\text{-}ucCST\_Pre}$ from $cone(\mathbf{m}')$.

From the previous property, all the paths of $\mathbf{S}$ subsumed by a path of $\mathbf{T}$ in Algorithm 8 can always be removed during the exploration. Unfortunately, it has been shown in section 4.5 that this step cannot be done in polynomial time. However, we can still define efficient heuristics to remove some of the paths from $\mathsf{k\text{-}ucCST\_Pre}(M, \mathbf{S})$ that are subsumed by paths of $\mathbf{S}$.

For example, consider the transition M defined by the guard $x_2 \geq 1 \wedge x_4 \geq 1$ and by the action $x_2' = x_2 - 1$, $x_4' = x_4 - 1$, $x_1' = x_1 + 1$ and $x_5' = x_5 + 1$. If we apply M to the upward-closed set with the minimal marking $\langle 0, 2, 1, 4, 0\rangle$, we obtain the upward closed set with the minimal marking $\langle 0, 3, 1, 5, 0\rangle$. Thus, the resulting marking does not yield new information and can be removed from $\mathbf{S}$ during the computation of $\mathsf{k\text{-}ucCST\_Pre}^*$. We will formalize this idea in the following proposition.

Proposition 15 Given a marking $\mathbf{m}$ and a transition M of Petri nets N, if for each x_i such that $d_i > 0$, $\mathbf{m}(x_i) \leq c_i$ holds, then it follows that $cone(\mathsf{k\text{-}ucCST\_Pre}(M, \mathbf{m})) \subseteq cone(\mathbf{m})$

When we compute $\mathsf{k\text{-}ucCST\_Pre}(M, \mathbf{S})$, it is interesting to efficiently detect and remove some paths of $\mathbf{S}$ satisfying Proposition 15 before the computation of $\mathsf{k\text{-}ucCST\_Pre}$. The following proposition gives a way to detect that all the paths of $\mathsf{k\text{-}ucCST\_Pre}(M, \mathbf{S})$ are subsumed by paths of $\mathbf{S}$.

Proposition 16 Let M be a transition of a Petri net N and $\mathbf{S}$ be a k-ucCST. If for all the layers i such that $d_i > 0$ we have that $val^{\mathbf{S}}(n) \leq c_i$ for all $n \in N_i^{\mathbf{S}}$, then $cones(\mathsf{k\text{-}ucCST\_Pre}(M, \mathbf{S})) \subseteq cones(\mathbf{S})$.

Checking condition of Proposition 16 can be done in linear time on k-ucCSTs and allows us to know with a low cost if it is interesting to compute $\mathsf{k\text{-}ucCST\_Pre}$ for a transition M on a k-ucCST $\mathbf{S}$.

When $cones(\mathsf{k\text{-}ucCST\_Pre}(M, \mathbf{S})) \not\subseteq cones(\mathbf{S})$, Algorithm 9 can be used to reduce $\mathbf{S}$ by removing edges e such that all the paths passing by e define upward-closed sets including their set of predecessors.

Algorithm 9: PrunePre

```
Input: S : k-ucCST, M : transition of Petri nets
begin
    ComputeSub↓(S, M);
    ComputeSub↑(S, M);
    Remove all edges e = ⟨n, m⟩ such that Sub↓(n) = false ∧ Sub↑(m) = false;
end
```

$\mathsf{ComputeSub}^{\downarrow}$ is defined as Algorithm 10 (we assume that $newval(n) = max(c_i, val(n) - d_i)$ where the guard on the variable corresponding to the node n is $x_i \geq c_i$ and the action $x_i := x_i + d_i$ i.e. $newval(n)$ is the new value of n when computing the pre-k-ucCST corresponding to k-ucCST_Pre$(M, \mathbf{S})$).

The idea behind Algorithm 10 is the following. In a top-down manner, the algorithm detects for the nodes n of the ith layer if n has a (not necessarily immediate) predecessor node whose value decreases when applying k-ucCST_Pre on $\mathbf{S}$ for the transition M. $Sub^{\downarrow}(n) = true$ if such a predecessor node exists for the node n, otherwise $Sub^{\downarrow}(n) = false$.

The function $\mathsf{ComputeSub}^{\uparrow}$ is defined from $\mathsf{ComputeSub}^{\downarrow}$ by traversing the k-ucCST from the last layer to the first one, replacing the relation $succ^{\mathbf{S}}$ by $father^{\mathbf{S}}$ and $Sub^{\downarrow}$ by $Sub^{\uparrow}$.

In that case, $Sub^{\downarrow}(n) = true$ if there exists a successor node whose value decreases when applying k-ucCST_Pre on $\mathbf{S}$ for the transition M, otherwise $Sub^{\downarrow}(n) = false$.

So, if $Sub^{\downarrow}(n) = false$ and $Sub^{\uparrow}(n') = false$ with $n' \in succ^{\mathbf{S}}(n)$, then all the paths passing through the edge $\langle n, n' \rangle$ correspond to paths defining upward closed sets including their set of predecessors for the transition M. Indeed, all the nodes encoding those paths have a value that does not decrease when applying M. Hence, such edges can be removed from the tree before computing k-ucCST_Pre$(M, \mathbf{S})$. This is what Algorithm 9 does: first, it computes $\mathsf{ComputeSub}^{\uparrow}$ and $\mathsf{ComputeSub}^{\downarrow}$ and then removes the useless edges with respect to the information given by those two functions. Notice that for the nodes n whose value decrements when applying M are such that $Sub^{\uparrow}(n) = true$ and $Sub^{\downarrow}(n) = true$. From Algorithm 10, it follows that all its (not necessarily immediate) predecessor nodes and successor nodes n' are such that, respectively, $Sub^{\uparrow}(n) = true$ and $Sub^{\downarrow}(n) = true$. As a consequence, if we apply twice Algorithm 9 on the k-ucCST $\mathbf{S}$ then the second application of the algorithm will not remove new edges.

Notice also that after applying Algorithm 9, some nodes n of $\mathbf{S}$ (different from the *root* and the *end* nodes) could have no direct predecessors ($father^{\mathbf{S}}(n) = \emptyset$) or direct successors ($succ^{\mathbf{S}}(n) = \emptyset$). Those nodes are useless since they do not appear into a path of $\mathbf{S}$, and hence can be removed. The useless nodes are removed by applying a bottom-up traversal of $\mathbf{S}$ to remove the nodes n with $succ^{\mathbf{S}}(n) = \emptyset$ and a top-down traversal of $\mathbf{S}$ to remove the nodes n with $father^{\mathbf{S}}(n) = \emptyset$.

The two following propositions hold.

Proposition 17 Given a transition M of a Petri net N, a k-ucCST $\mathbf{S}$ and the pre-k-ucCST $\mathbf{S}'$ obtained by applying PrunePre on $\mathbf{S}$ with M and then removing the useless nodes, we have

$$\text{k-ucCST\_Pre}(cones(\mathbf{S})) = \text{k-ucCST\_Pre}(cones(\mathbf{S}')).$$

Proposition 18 The complexity of PrunePre is linear in the size of the input k-ucCST.

Proof. Stems from the linear complexity of the breath-first search. □

Another way to compute the k-ucCST_Pre operation is to follow the idea given in [Zam97] by defining a k-sharing tree $\mathbf{S}_{rel}$, called transition tree, describing the set of transitions. The value of each node of the transition tree is a couple where the first component describes the guard of a transition (i.e. a constant c_i), the second component describes the effect of a transition (i.e. a constant d_i) and each path corresponds to a transition. Figure 5.5 shows the transition tree for the transitions M_1 with guard $x_3 \geq 1$ and effect $x_3 := x_3 - 1$, M_2 with guard $x_3 \geq 1$ and effect $x_3 := x_3 - 1 \wedge x_4 := x_4 + 1$, M_3 with guard $x_2 \geq 1 \wedge x_3 \geq 1$ and

Algorithm 10: ComputeSub$^{\downarrow}$

Input: $\mathbf{S}$: k-ucCST, M : transition of Petri nets
begin
  foreach $n \in N^{\mathbf{S}}$ **do**
    $Sub^{\downarrow}(n) \leftarrow false$;
  foreach $i : 1$ *to* k (*top to bottom*) **do**
    foreach *node* $n \in N_i^{\mathbf{S}}$ **do**
      if $newval(n) < val^{\mathbf{S}}(n)$ **then**
        $Sub^{\downarrow}(n) \leftarrow true$;
      if $Sub^{\downarrow}(n) = true$ **then**
        foreach $n' \in succ^{\mathbf{S}}(n)$ **do**
          $Sub^{\downarrow}(n') \leftarrow true$;
end

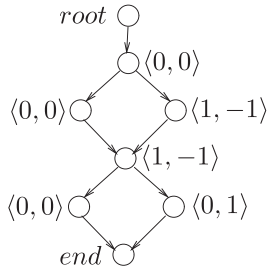

Figure 5.5: Example of transition tree.

effect $x_2 := x_2 - 1 \wedge x_3 := x3 - 1$ and the transition M_4 with guard $x_2 \geq 1 \wedge x_3 \geq 1$ and effect $x_2 := x_2 - 1 \wedge x_3 := x3 - 1 \wedge x_4 := x_4 + 1$.

By computing the operation corresponding to the product of automata between the transition tree $\mathbf{S}_{rel}$ and a k-ucCST $\mathbf{S}$,we compute $\mathbf{S}_{pre}$, equivalent to k-ucCST_Pre($\mathbf{S}$), in one step (i.e. according to all the transitions). More precisely, this operation works as follows. For any node n_1 of the *ith* layer of $\mathbf{S}$ and any node n_2 of the *ith* layer of the transition tree, we construct a node $\langle n_1, n_2 \rangle$ in the *ith* layer of the resulting k-ucCST $\mathbf{S}_{pre}$. In $\mathbf{S}_{pre}$, we will have $\langle n'_1, n'_2 \rangle \in succ^{\mathbf{S}_{pre}}(\langle n_1, n_2 \rangle)$ if and only if $n'_1 \in succ^{\mathbf{S}}(n_1)$ and $n'_2 \in succ^{\mathbf{S}_{rel}}(n_2)$.

For a node $\langle n_1, n_2 \rangle$ of $\mathbf{S}_{pre}$, we define $val^{\mathbf{S}_{pre}}(\langle n_1, n_2 \rangle) = max(c_i, val^{\mathbf{S}}(n_1) - d_i)$ where c_i is the constant defining the guard and d_i is the constant defining the effect in n_2.

So, each path $\langle\langle \top^{\mathbf{S}}, \top^{\mathbf{S}_{rel}} \rangle, \langle n_1, n'_1 \rangle, \ldots, \langle n_k, n'_k \rangle, \langle \bot^{\mathbf{S}}, \bot^{\mathbf{S}_{rel}} \rangle\rangle$ of $\mathbf{S}_{pre}$ corresponds to the path $p_1 = \langle \top^{\mathbf{S}}, n_1, \ldots, n_k, \bot^{\mathbf{S}} \rangle$ of $\mathbf{S}$ and a path $p_2 = \langle \top^{\mathbf{S}_{rel}}, n'_1, \ldots, n'_k, \bot^{\mathbf{S}_{rel}} \rangle$ of $\mathbf{S}_{rel}$, and define the result of applying the transition M corresponding to p_2 on the minimal marking defined by p_1. This product can be done in polynomial time but the resulting pre-k-ucCST could violate the conditions 5-6. The algorithms of Figure 27 and Figure 28 must be then applied to obtain a k-ucCST.

That approach is in general more efficient than successively compute the pre operation for each transition.

The k-ucCST-based Backward Search Engine

We are ready now to formulate our k-ucCST-backward reachability algorithm.

Algorithm 11: $\mathsf{Symbolic}_{\mathsf{BS}}$

```
Input: ⟨P, B⟩ : Petri net, S_U : k-ucCST, m_0 ∈ (ℕ ∪ {+∞})^k
Result: boolean
begin
    S ← S_U;
    T ← k-ucCST_Empty_set;
    while not(k-ucCST_Subsumption(S, T)) do
        T ← k-ucCST_Union(T, S);
        R := k-ucCST_Empty_set;
        foreach M ∈ B do
            N ← k-ucCST_Pre(S, M);
            k-ucCST_Heuristics_Minimisation(N, R, T);
            k-ucCST_Heuristic_Normal_Form(N);
            R ← k-ucCST_Union(N, R);
        S ← R;
    return k-ucCST_Sat(m_0, T);
end
```

The algorithm works as follows (see Algorithm 11). First of all, we store intermediate results in two variables. The variable $\mathbf{S}$ stores the current *frontier*, i.e. the set of minimal elements that have not been treated yet, of the breadth-first search performed by the algorithm. The variable $\mathbf{T}$ stores the set of visited generators.

Before entering the main loop, we test subsumption between $\mathbf{S}$ and $\mathbf{T}$. For this purpose, the following heuristics seems to work fine in practice.

1. We first compute the forward and backward simulation relations between the nodes of $\mathbf{S}$ and the nodes of $\mathbf{T}$.

2. If the root of $\mathbf{S}$ is forward simulated by the root of $\mathbf{T}$ or if the end node of $\mathbf{S}$ is backward simulated by the end node of $\mathbf{T}$, then we know that all the generators of $\mathbf{S}$ are subsumed by some generators of $\mathbf{T}$, thus the fixpoint is reached.

3. If the test fails, we perform a depth-first, top-down visit of the k-ucCST $\mathbf{S}$ in order to compare its tuples with those of $\mathbf{T}$. During the depth-first visit, we use however the information previously computed via the forward simulation as follows. Each time we reach a node n that is forward simulated by a node of $\mathbf{T}$, we stop the exploration: all the elements in the subtree rooted at n will be subsumed by elements of $\mathbf{T}$.

In the main loop, we compute the new frontier $\mathbf{N}$ transition-by-transition via the symbolic operator k-ucCST_Pre. As a further heuristics to keep the size of $\mathbf{N}$ small, after computing k-ucCST_Pre, we apply simulation-based heuristics to remove *redundant* generators from the frontier. The function k-ucCST_Heuristics_Minimisation($\mathbf{N}, \mathbf{R}, \mathbf{T}$) uses simulation relations between nodes of $\mathbf{N}$ and $\mathbf{R}$, and the nodes of $\mathbf{N}$ and $\mathbf{T}$ as presented in Section 4.5.8 to approximate the minimisation operation between $\mathbf{N}$ and $\mathbf{R}$ (to remove paths of $\mathbf{N}$), $\mathbf{R}$ and $\mathbf{N}$ (to remove paths of $\mathbf{R}$), $\mathbf{N}$ and $\mathbf{T}$ (to remove paths of $\mathbf{N}$), and $\mathbf{T}$ and $\mathbf{N}$ (to removed paths of $\mathbf{T}$). The function k-ucCST_Heuristic_Normal_Form($\mathbf{N}$) uses simulation relations of nodes of $\mathbf{N}$ to remove redundancies (and approximate the normal form).

Algorithm 12: Symbolic$_{\mathsf{BS}}$

```
Input: ⟨P, B⟩ : Petri net, S_U : k-ucCST, m_0 ∈ (ℕ ∪ {+∞})^k
Result: boolean
begin
    S ← S_U;
    T ← k-ucCST_Empty_set;
    while not(k-ucCST_Empty(S)) do
        T ← k-ucCST_Union(T, S);
        R := k-ucCST_Empty_set;
        foreach M ∈ B do
            N ← k-ucCST_Pre(S, M);
            k-ucCST_Minimization(N, R);
            k-ucCST_Minimization(R, N);
            k-ucCST_Minimization(N, T);
            k-ucCST_Minimization(T, N);
            k-ucCST_NormalForm(N);
            R ← k-ucCST_Union(N, R);
        S ← R;
    return k-ucCST_Sat(m_0, T);
end
```

An other way consists to compute the different operations on k-ucCSTs by using the algorithms presented in Section 4.5 (see Algorithm 12). In that case, k-ucCSTs are in normal form and the k-ucCST $\mathbf{S}$ has only paths that are not subsumed by the paths of $\mathbf{T}$. More precisely, the subsumption test is replaced by the emptiness test on $\mathbf{S}$, k-ucCST_Heuristics_Minimisation is replaced by applications of k-ucCST_Minimization (see Algorithm 5) between the k-ucCSTs $\mathbf{N}$ and $\mathbf{R}$, and $\mathbf{N}$ and $\mathbf{T}$ and the operation k-ucCST_Heuristic_Normal_Form is replaced by the operation k-ucCST_NormalForm (see Algorithm 3) to put k-ucCSTs in normal form. Notice that since the k-ucCST_Minimization operation is applied between the k-ucCSTs manipulated by Algorithm 12, k-ucCST_Union returns an irredundant k-ucCST even if it is computed by using the sharing tree operation union (see Table 4.1) as proposed in Section 4.5.5.

Those two approach will be evaluated in Section 5.5.

Preliminary Experiments

Let us go back to the example of the Manufacturing System discussed in Section 5.2.1. The use of k-ucCSTs as the assertional language to represent intermediate results dramatically speeds up the analysis of the verification problem proposed in [DEP99]. The computation

Example	NV	NR	NS	LR	FSR	ET	NE	Nodes	Ratio
Man. Sys.	13	6	24	no	__	39s	7563	4420	0.04
Man. Sys.	13	6	24	yes	5	68s	727	1772	0.18

Table 5.3: NV=No. Variable; NR=No. Rules; NS=No Steps; LR=Use of local reduction; FSR=Period in the use of sim. reduction (__=not used); ET=Execution Time; NE=No. of Elem. of the result k-ucCST; Nodes = No of nodes of the resulting k-ucCST;Ratio=Nodes/(NV*NE).

based on the k-ucCST-search engine of Figure 11 terminates after 39 seconds on a Sun Ultra Sparc (see Table 5.3). In a first experiment we have not removed the redundancies from [4] $\mathbf{S}_{\mathsf{pre}^i}$, whereas in a second experiment we have applied the reduction based on the forward simulation (every 5 iterations of the *while* loop of Algorithm 11) (see Table 5.3, column FSR). Simulation-based reduction (every 5 iterations of the *while* loop of Algorithm 11) allows us to reduce the set of generators by a factor of ten (note: removing *all* redundancies yields 450 elements).

Finally, the ratio (see Table 5.3, column Ratio) allows us to compare the number of nodes of the k-ucCST at the end of the computation with the number of nodes that we would have if we had no sharing of prefixes and suffixes of paths. As shown on the table, k-ucCSTs are compact with respect to the number of minimal elements of the upward closed sets they represent.

Notice that, on this example, although the numbers of minimal elements represented by paths of k-ucCSTs are significantly reduced by using simulation relations, the size of k-ucCSTs does not decrease with the same ratio. This is a characteristic of symbolic techniques where the size of the data structures do not depend on the size of the sets they represent. As a consequence, on this particular example, the cost of the simulation relation heuristics is bigger than the gain we obtain by manipulating more compact structures and the execution time obtained in the first experiment is smaller than the execution time obtained in the second one. We will see however in Section 5.5 that, most of the time, simulation relations are useful in practice.

5.2.2 Symbolic Backward Reachability Algorithm for MTNs

A symbolic backward algorithm for MTNs is obtained from Algorithm 11 by defining a new algorithm to compute the operation k-ucCST_Pre on MTNs instead of on basic Petri nets.

However, the backward effect of (bounded or not) transfers are not trivial to compute. Actually, it corresponds to non-deterministic moving of tokens (see example 4).

Example 4 *Consider the transfer that moves all the tokens from the place p_5 into the place p_2, defined through the equation $x_2' = x_2 + x_5, x_5' = 0$ (as usual, primed versions of the variables describe the values of variables after transformation). If we have no tokens in the place p_5 and two tokens in the place p_2 after the transfer (see Figure 5.6(a)), each of these tokens could be either in the place p_2 or in the place p_5 before the transfer. More precisely, the three possible configurations before the transfer are :*

- *the two tokens were already in the place p_2, see Figure 5.6(a), and no token moves in this case when firing the multi-transfer;*

[4] $\mathbf{S}_{\mathsf{pre}^i}$ is the k-ucCST $\mathbf{S}$ obtained after iterating i times the *while* loop of Algorithm 11.

- *one token was in the place p_2 and the second one was in the place p_5, see Figure 5.6(b). In this case, only the token in the place p_5 moves into the place p_2 when firing the multi-transfer;*
- *the two tokens were in the place p_5, see Figure 5.6(c), and both tokens are transfered into the place p_2 when firing the multi-transfer.*

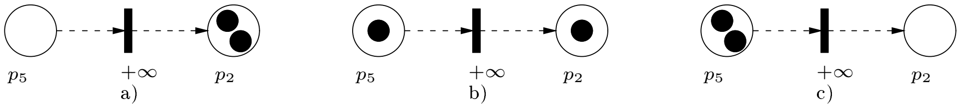

Figure 5.6: Backward effect of a transfer of tokens from the places p_2 to p_5

The difficulty of the computation of $\mathsf{k\text{-}ucCST\_Pre}$ on MTNs appears when multi-transfers contain different transfer blocks. All the effects of each of them must be combined together. To avoid this exponential explosion, we decompose the computation of $\mathsf{k\text{-}ucCST\_Pre}$ for a general multi-transfer into the computation of more simple operations. The results of the different computations are then put together to finally obtain the desired k-ucCST.

In the following, we consider that $\mathcal{I}_0$ and $\mathcal{O}_0$ are defined such that $\mathcal{I}_0(p) = 0$ and $\mathcal{O}_0(p) = 0$ for all the places p.

Following Condition 2 of Definition 9, the transfer blocks of a multi-transfer involve distinct sets of places. It follows that we can compute the pre operation for a multi-transfer by computing the effect of each of its transfer blocks independently, as stated by the following lemma.

Lemma 13 (Decomposition of pre) Let U be a set of markings (not necessarily upward closed) and M be a multi-transfer of the form $\langle\langle\mathcal{I},\mathcal{O}\rangle,\{S_1,\dots S_n\}\rangle$. Let $M_i (1 \leq i \leq n)$ be the multi-transfer $\langle\langle\mathcal{I}_0,\mathcal{O}_0\rangle,\{S_i\}\rangle$, $M_{\mathcal{I}}$ be the multi-transfer $\langle\langle\mathcal{I},\mathcal{O}_0\rangle,\emptyset\rangle$ and $M_{\mathcal{O}}$ be the multi-transfer $\langle\langle\mathcal{I}_0,\mathcal{O}\rangle,\emptyset\rangle$.

Then, the following holds.

$$\mathsf{pre}_M(U) = \mathsf{pre}_{M_{\mathcal{I}}} \circ \mathsf{pre}_{M_1} \circ \mathsf{pre}_{M_2} \circ \cdots \circ \mathsf{pre}_{M_n} \circ \mathsf{pre}_{M_{\mathcal{O}}}(U)$$

where $\circ$ is the composition operator.

Notice that the order of the computation of the pre operation for $M_1,\dots,M_n$ is not relevant.

To compute the backward effect of one transfer block with the bound $+\infty$, we can proceed following the same way. Indeed, the transfers of a MTN involve distinct set of places each other following Condition 2 of Definition 9. So we can compute the backward effect of the transfers of a transfer block with the bound $+\infty$ independently.

Lemma 14 (Decomposition of pre for a transfer block with the bound $+\infty$) Let U be a set of markings (not necessarily upward closed) and M be a multi-transfer of the form $\langle\langle\mathcal{I}_0,\mathcal{O}_0\rangle,\langle\{B_1,\dots,B_n\},+\infty\rangle\rangle$. Let $M_i = \langle\langle\mathcal{I}_0,\mathcal{O}_0\rangle,\langle\{B_i\},+\infty\rangle\rangle$ for i such that $1 \leq i \leq n$.

Then, the following holds.

$$\mathsf{pre}_M(U) = \mathsf{pre}_{M_1} \circ \mathsf{pre}_{M_2} \circ \cdots \circ \mathsf{pre}_{M_n}(U)$$

where $\circ$ is the composition operator.

Notice again that the order of computation of $M_1, \ldots, M_n$ is not relevant.

We now state how to compute the backward effect of a transfer block with the bound 1. The computation of pre for transfer blocks with a bound $n \in \mathbb{N}$ is discussed afterwards. Let us first define $\mathsf{pre}^{\leftarrow}$ as follows.

Given a set U (not necessarily upward closed) and a multi-transfer $M_B = \langle\langle \mathcal{I}_0, \mathcal{O}_0\rangle, \langle\{B\},1\rangle\rangle$, i.e. corresponding to the transfer $B = \langle P, p\rangle$ with the bound 1, we define $\mathsf{pre}^{\leftarrow}_{M_B}(U)$ as follows.

$$\begin{array}{ll}\mathsf{pre}^{\leftarrow}_{M_B}(U) = & \{\mathbf{m} | \exists \mathbf{m}' \in U : \mathbf{m}'(p) > 1 \text{ and } \mathbf{m}(p) = \mathbf{m}'(p) - 1 \text{ and} \\ & \exists p' \in P : \mathbf{m}(p') = \mathbf{m}'(p') + 1 \text{ and } \forall p'' \notin \{p, p'\} : \mathbf{m}(p'') = \mathbf{m}'(p'')\}\end{array}$$

Intuitively, $\mathsf{pre}^{\leftarrow}_{M_B}(U)$ is the set of markings from which U is reached by transferring (exactly) one token from P to p.

Lemma 15 (Decomposition of pre for a transfer block with the bound 1) Let U be a set of markings and M be a multi-transfer of the form $\langle\langle \mathcal{I}_0, \mathcal{O}_0\rangle, \langle\{B_1, \ldots, B_n\}, 1\rangle\rangle$. Let $M_i (1 \le i \le n)$ be the multi-transfer $\langle\langle \mathcal{I}_0, \mathcal{O}_0\rangle, \langle\{B_i\}, 1\rangle\rangle$.

Then, the following holds.

$$\mathsf{pre}_M(U) = U_0 \cup (\bigcup_{i=1}^{n} \mathsf{pre}^{\leftarrow}_{M_i}(U))$$

where $U_0 = \{\mathbf{m} \in U | \mathbf{m}(\bigcup_{i=1..n} P_i) = 0\}$ where P_i is the set of source places of B_i.

where U_0 is the set of markings from which M has no effect (since there is no tokens in the source places of the transfers of M).

In the case of transfer blocks with a bound $n \in \mathbb{N}$, we proceed as follows. From Definition 9, we have that all the transfers of a multi-transfer involve distinct sets of places and there is no cyclic transfer. It follows that instead of moving at most n tokens in one step, we can move them in n steps, by applying n times the pre operation for the transfer block with the bound 1.

To give the intuition behind our approach, consider the multi-transfer M_1 on the places p_1 and p_2 shown in Figure 5.7(a). Assume that M_1 is isolated in the sense that there is no other multi-transfer. Then, firing the multi-transfer M_1, that has one transfer block with the bound 2 (see Figure 5.7(a)), has the same effect than firing the sequence of (isolated) multi-transfers $M_{11}M_{12}$ of Figure 5.7(b) where the transfer blocks have the bound 1.

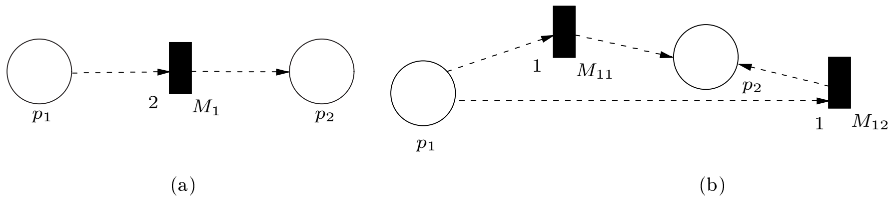

Figure 5.7: Simulation of a transfer block with bound 2 by two transfer blocks with bound 1.

So, our decomposition lemma in the case of transfer block with the bound $n \in \mathbb{N}$ is the following.

Lemma 16 (Decomposition of pre for a transfer block with a natural bound) Let U be a set of markings (not necessarily upward closed) and M be a multi-transfer of the form $\langle\langle\mathcal{I}_0, \mathcal{O}_0\rangle, \langle\{B_1, \ldots, B_n\}, n\rangle\rangle$ with $n \in \mathbb{N}$. Let $M' = \langle\langle\mathcal{I}_0, \mathcal{O}_0\rangle, \langle\{B_1, \ldots, B_n\}, 1\rangle\rangle$.

Then, the following holds.

$$\mathsf{pre}_M(U) = \mathsf{pre}^n_{M'}(U)$$

where $\mathsf{pre}^n_{M'}(U)$ corresponds to apply n times $\mathsf{pre}_{M'}$ from U.

Thus to compute the pre operation for a transfer block S with the bound n, we apply n times the pre operation on a transfer block similar to S but with the bound 1.

Algorithm to compute k-ucCST_Pre on MTNs

We present in this section symbolic algorithms to compute the k-ucCST_Pre operation on MTNs. As in the case of set operations and of the k-ucCST_Pre on basic Petri nets, symbolic means here that these algorithms do not work on the paths of k-ucCSTs but directly manipulate the graph structure of those k-ucCSTs.

Let us first focus on the set of predecessors of an upward-closed set of markings with one minimal element $\mathbf{m}$ for a multi-transfer of the form $M = \langle\langle\mathcal{I}_0, \mathcal{O}_0\rangle, \langle\{\langle P, p_t\rangle\}, +\infty\rangle\rangle$ i.e. with only one transfer B and the bound $+\infty$. Intuitively,

1. The operation must take into account the following postcondition (induced by the semantics of transfer blocks with bound $+\infty$): *all the source places in P must contain zero tokens after firing a* transfer block *with bound* $+\infty$
2. Applying pre on $cone(\mathbf{m})$ for M, we should obtain a representation of the upward-closed set U_{pre} whose markings correspond to some possible distribution of tokens *before* the transfer of the tokens from the source places to the target place (see example 4).

Formally, we have the following.

Proposition 19 Let M be a multi-transfer of the form $\langle\langle\mathcal{I}_0, \mathcal{O}_0\rangle, \langle\{\langle P, p\rangle\}, +\infty\rangle\rangle$, i.e. corresponding to the transfer $\langle P, p\rangle$ with the bound $+\infty$. The predecessor set of $cone(\mathbf{m})$ is the upward closed set $\mathsf{pre}(cone(\mathbf{m}))$ such that

1. if $\mathbf{m}(P) > 0$, $\mathsf{pre}(cone(\mathbf{m})) = \emptyset$;
2. if $\mathbf{m}(P) = 0$, we have $min(\mathsf{pre}(cone(\mathbf{m}))) = \{\mathbf{m}' | \forall p' \notin P \cup \{p\} : \mathbf{m}'(p') = \mathbf{m}(p')$ and $\mathbf{m}'(P \cup \{p\}) = \mathbf{m}(p)\}$.

Removing from an k-ucCST $\mathbf{S}$ the paths corresponding to the minimal markings with tokens in the source places of a transfer, i.e. corresponding to case 1 of proposition 19, can be done in time polynomial by first removing the nodes n of $\mathbf{S}$ such that $val^{\mathbf{S}}(n) \neq 0$ from the layers corresponding to the source places. After that treatment, some nodes n of $\mathbf{S}$ could have no direct predecessors $father^{\mathbf{S}}(n) = \emptyset$ or no direct successors $succ^{\mathbf{S}}(n) = \emptyset$ and thus are useless. Those useless nodes are removed and then Algorithm 28 is applied to ensure condition 6 of definition 42.

To manage the case 2 of Proposition 19, we need Algorithm 14. Given a k-ucCST $\mathbf{S}$, a number of tokens nb_tokens and a set of layers S, the function Add_Tokens computes the k-ucCST $\mathbf{S}$' such that each path of $\mathbf{S}$' corresponds to a minimal element of $\mathbf{S}$ where

nb_tokens have been added in the places corresponding to the layers in S. This algorithm works recursively on the sub-trees of the k-ucCST by using the recursive function Put_Tokens. Put_Tokens distributes an amount of tokens into a sub tree of $\mathbf{S}$. For each successor node n of the root node of the current subtree, the following holds. If the layer of n is in S, then Put_Tokens adds i tokens into n (adds i to the value of n) and a recursive call is applied to distribute the remaining tokens in the next layers (lines 5-8 of Algorithm 13). Such a recursive call is applied for all the possible amounts of tokens that could be added into n, i.e. if the number of tokens that remains to distribute is k then Put_Tokens adds $0, 1, \dots, k$ tokens into n and apply recursive calls to distribute $k, k-1, \dots, 0$ tokens in the rest of the layers.

If the layer of n is not in S, then a recursive call is applied without adding tokens in n (lines 9 of Algorithm 13). A traversal of the k-ucCST is validated (lines 1-4 of Algorithm 13), i.e. the path corresponding to the current traversal of the k-ucCST will appear in the set of paths of the resulting k-ucCST, if all the tokens specified to the function Add_Tokens have been distributed among the layers in S during the traversal.

Put_Tokens could distribute the same amount of tokens in the same sub-tree several times, i.e. during the traversal of the k-ucCST, Put_Tokens could be applied several times with the same parameters. For this reason, a cache (that is empty at the beginning of the algorithm) is used to cache the result of distributing i tokens into a sub-tree rooted by a node n. The cache is accessed using the memo? and memo! functions. For a node n and a number of tokens nb_tokens, memo!$(n, nb\_tokens, result)$ stores the subtree rooted by the $result$ node that corresponds to the sub-tree rooted by n where nb_tokens have been added into the layers of S. Given a node n and a number of tokens nb_tokens, memo?(n, nb_tokens) returns the sub-tree corresponding to the subtree rooted by n where nb_tokens have been added into layers in S. If no such sub-tree have been cached, then memo? returns a *notmemorized* value.

Before each recursive call of Put_Tokens for the node n and the value nb_tokens, memo? is called to check if the result has already been computed. If it is the case, memo? returns the node that would be computed by the recursive call and Put_Tokens is not applied. Otherwise, if the *notmemorized* value is returned by memo?, Put_Tokens is applied but the result will be cached and the function will be not applied again with the same parameters.

More precisely, the algorithm that we propose to compute the predecessors of a k-ucCST $\mathbf{S}$ for a transfer with the bound $+\infty$ works as follows (see Algorithm 15). First, all the paths corresponding to minimal markings of $\mathbf{S}$ with at least one token in the source places are removed from $\mathbf{S}$ (line 1 of Algorithm 15). That corresponds to case 1 of proposition 19. Case 2 of proposition 19 is handled on k-ucCSTs by making a distinct treatment for each possible value encountered in the layer corresponding to the target place of the transfer. For a value c in the target layer, a k-ucCST $\mathbf{S}_c$ containing only the remaining minimal elements of $\mathbf{S}$ with c tokens in the target place is constructed (line 3 of Algorithm 15). Then, c tokens are removed from the target place, i.e. the value of the node of the corresponding layer is set to 0 (line 4-5 of Algorithm 15), and finally, Add_Tokens is applied on $\mathbf{S}_c$ for c tokens (line 6 of Algorithm 15). The set of layers given as parameter is composed of the layers corresponding to the target place and the source places of the transfer. Add_Tokens distributes c tokens among this set of layers in $\mathbf{S}_c$. This corresponds to compute all the possible distributions of the tokens before the transfer.

The following example will help in understanding the algorithm.

Example 5 *Consider the k-ucCST* $\mathbf{S}$ *in Figure 5.8(a) encoding the upward closed set of mark-*

Algorithm 13: Put_Tokens

```
Input: nb_tokens : integer, n : node, S : set of layers
Result: a node
begin
    if val(n) = ⊥ then
        if nb_tokens = 0 then
            result = memo?(n, 0) ;
            if result = notmemorized then
                Construct a new node result with val(result) = ⊥ ;
        else
            result = null ;
    else
        Construct a new node result with val(result) = val(n) ;
        foreach Son ∈ succ^S(n) do
            if layer of Son is in S then
                foreach i ∈ [0, nb_tokens] do
                    s' = memo?(Son, nb_tokens − i) ;
                    if s' = notmemorized then
                        s' = Put_Tokens(nb_tokens − i, Son, S) ;
                    if s' ≠ null then
                        Construct a new node Son' with val(Son') = val(s') + i and
                        succ^S'(Son') = succ^S'(s') ;
                        succ^S'(result) = succ^S'(result) ∪ {Son'} ;
            else
                Son' = memo?(Son, nb_tokens) ;
                if Son' = notmemorized then
                    Son' = Put_Tokens(nb_tokens, Son, S) ;
                if Son' ≠ null then
                    succ^S'(result) = succ^S'(result) ∪ {Son'} ;
    if succ^S'(result) = ∅ then
        result = null ;
    memo!(n, nb_tokens, result) ;
    return result ;
end
```

Algorithm 14: Add_Tokens

Input: $\mathbf{S}$: k-ucCST, nb_tokens : integer, S : set of layers
Result: k-ucCST
Let $\mathbf{S}$' be the k-ucCST built as follows ;
$root^{\mathbf{S}'} \leftarrow \mathsf{Put\_Tokens}(nb\_tokens, root^{\mathbf{S}}, S)$;
return $\mathbf{S}$' ;

Algorithm 15: k-ucCST_Pre

Input: $\mathbf{S}$: k-ucCST; $\langle\langle \mathcal{I}_0, \mathcal{O}_0\rangle, \{\langle P, p\rangle\}, \infty\rangle$: multi-transfer
Result: k-ucCST
begin
  Let $\mathbf{S}'$ such that $elem(\mathbf{S}') = \{\mathbf{m} \in elem(\mathbf{S}) : \mathbf{m}(P) = 0\}$;
  Apply Algorithm 28 that ensures Condition 6 on $\mathbf{S}'$;
  $\mathbf{R} \leftarrow$ k-ucCST_Empty_set;
  foreach *value c in the layer associated to place p* **do**
    $\mathbf{S_c} \leftarrow \mathbf{S}'$ where the nodes n in the layer of p with $val(n) \neq c$ have been removed;
    foreach *node n of the layer in $\mathbf{S}_c$ associated to p* **do**
      $val(n) \leftarrow 0$;
    $\mathbf{T}_c \leftarrow \mathsf{Add\_Tokens}(\mathbf{S}_c, c, P \cup \{p\})$;
    Apply Algorithm 27 that ensures Condition 5 on $\mathbf{T}_c$;
    Apply Algorithm 28 that ensures Condition 6 on $\mathbf{T}_c$;
    $\mathbf{R} \leftarrow \mathsf{k\text{-}ucCST\_Union}(\mathbf{R}, \mathbf{T}_c)$;
  return $\mathbf{R}$;
end

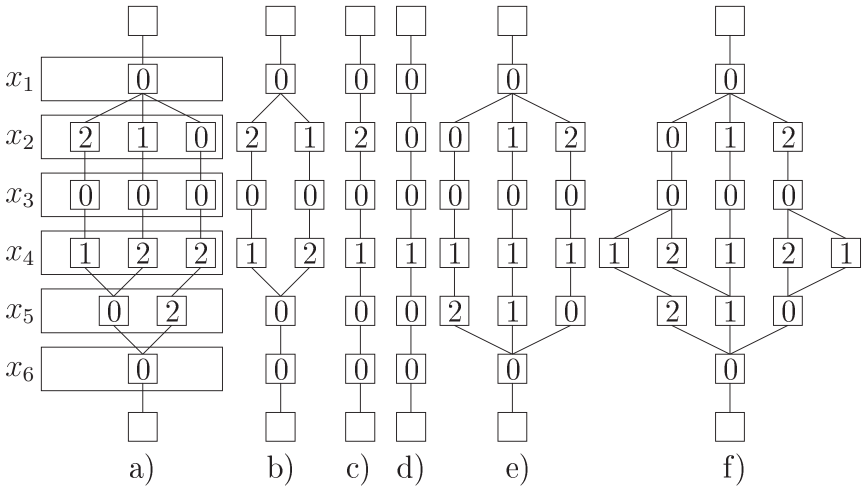

Figure 5.8: k-ucCSTs generated during the computation of the transfer $x_2' = x_2 + x_5, x_5' = 0$

ings with the minimal elements $\langle 0,2,0,1,0,0\rangle$, $\langle 0,1,0,2,0,0\rangle$ and $\langle 0,0,0,2,2,0\rangle$ corresponding to the formula $\Phi = (x_2 \geq 2 \wedge x_4 \geq 1) \vee (x_2 \geq 1 \wedge x_4 \geq 2) \vee (x_4 \geq 2 \wedge x_5 \geq 2)$. Consider again the transfer of tokens from the place p_5 into the place p_2 characterized by the expression $x_2' = x_2 + x_5, x_5' = 0$. When applied on $\mathbf{S}$, Algorithm 15 performs the steps shown in 5.8(b-f). Specifically, it first computes (b) by removing the tuple $\langle 0,0,0,2,2,0\rangle$ that does not satisfy $x_5' = 0$ (line 1 of Algorithm 15). The k-ucCST $\mathbf{S}_c$, for $c = 2$, (line 3 of Algorithm 15) computed at the second iteration of the loop (line 2 of Algorithm 15) is shown in (c). This k-ucCST $\mathbf{S}_c$ contains all the minimal elements of $\mathbf{S}$ that has c tokens in the target of the transfer, i.e. where p contains c tokens. From this k-ucCST, we obtain (d) by removing the constraints on the second variable by replacing the value of the nodes of the second layer (corresponding to the target place) by 0 (lines 4-5 of Algorithm 15). The k-ucCST resulting from the application of the function Add_Tokens *on the k-ucCST (d) for the set of places $\{p_2, p_5\}$ and the constant 2 is shows in (e) (line 6 of Algorithm 15). Intuitively, those two steps consist to compute all the possible configurations of the c tokens before the application of the transfer and for all the minimal elements of $\mathbf{S}_c$, i.e.* Add_Tokens *computes the predecessor set of the minimal elements in $\mathbf{S}$ that have c tokens in p after the transfer. Finally, doing the same for $\mathbf{S}_c$ with $c = 1$, the result of Algorithm 15 when applied on $\mathbf{S}$ is the k-ucCST (f).*

To compute the pre operation of a transfer block with the bound 1, as stated by Lemma 15, we need to compute the operation k-ucCST_Pre$^{\leftarrow}$ (the k-ucCST_Union operation have been discussed in Section 4.5). k-ucCST_Pre$^{\leftarrow}$ is computed on a k-ucCST $\mathbf{S}$ for a transfer $\langle P, p\rangle$ as follows (see Algorithm 16). Intuitively, k-ucCST_Pre$^{\leftarrow}$ computes the predecessor markings of $cones(\mathbf{S})$ that reaches $cones(\mathbf{S})$ by moving one token from the source places P to the target place p. As a consequence, we are only interested in the markings of $cones(\mathbf{S})$ that have at least one token in p. Indeed, the other markings are not reachable by moving one token to p.

First, Algorithm 16 removes the markings that have no tokens in p from $cones(\mathbf{S})$. This corresponds to replace all the values of nodes equals to 0 by 1 in the layer corresponding to p (lines 1-3 of Algorithm 16). Then, as k-ucCST_Pre$^{\leftarrow}$ computes the predecessors of $cones(\mathbf{S})$ that reach $cones(\mathbf{S})$ by moving one token from P to p, Algorithm 16 removes one tokens from the nodes in the layer corresponding to p and adds it into a node corresponding to a place in

Algorithm 16: k-ucCST_Pre$^{\leftarrow}$

Input: $\mathbf{S}$: k-ucCST; $\langle\langle \mathcal{I}_0, \mathcal{O}_0\rangle, \{\langle P, p\rangle\}, 1\rangle$: multi-transfer
Result: k-ucCST
begin
  $\mathbf{S}' \leftarrow \mathsf{Copy}_{\text{k-ucCST}}(\mathbf{S})$;
1 **foreach** *node n of the layer of $\mathbf{S}'$ associated to p* **do**
2   **if** $val(n) = 0$ **then**
3     $val(n) \leftarrow 1$;
4 **foreach** *node n of the layer of $\mathbf{S}'$ associated to p* **do**
5   $val(n) \leftarrow val(n) - 1$;
6 $\mathbf{R} \leftarrow \mathsf{Add\_Tokens}(\mathbf{S}', 1, P)$;
  Apply algorithm 27 that ensure Conditions 5 on $\mathbf{R}$;
  Apply algorithm 28 that ensure Conditions 6 on $\mathbf{R}$;
  return $\mathbf{R}$;
end

P (this corresponds to the backward effect of moving one token from P to p).

More precisely, one token is removed from the target place by decrementing the value of all the nodes in the corresponding layer (lines 4-5 of Algorithm 16). Then, Add_Tokens is applied on the resulting k-ucCST with the constant 1 and the set of layers corresponding to the places in P (the source places). Add_Tokens returns the k-ucCST $\mathbf{R}$ where one token have been added in the source places P i.e., it returns the k-ucCST corresponding to the set of markings from which $cones(\mathbf{S})$ is reachable by moving one token from P to p. Finally, Algorithm 27 and Algorithm 28 presented in Appendix are applied to ensure condition 5-6 of definition 42.

The set of predecessors of a k-ucCST $\mathbf{S}$ for a transfer block with the bound 1 is computed by first computing the k-ucCST $\mathbf{S}_0$ containing all the paths of $\mathbf{S}$ where there is no tokens in the source places of the transfers of the block. Those paths correspond to markings on which the transfer block has no effect. This is computed from a copy of $\mathbf{S}$ by first removing the nodes n of the layers corresponding to the source places such that $val^{\mathbf{S}}(n) \neq 0$ and then apply Algorithm 28 to ensure the maximum suffix sharing. Applying this treatment, we remove all the paths from the copy of $\mathbf{S}$ corresponding to tuples where there is at least one node corresponding to a source place that has a value different from 0, i.e. the tuples from which the transfer block will move one token from the source places to the target place. Finally, the k-ucCST $\mathbf{S}_{\mathsf{pre}}$ describing the set of predecessors is obtained by making the disjunction between $\mathbf{S}_0$ and the results of the application of k-ucCST_Pre$^{\leftarrow}$ on $\mathbf{S}$ for each transfers of the block, as stated by Lemma 15. As $\mathbf{S}_{\mathsf{pre}}$ is not guaranteed to be irredundant, even when $\mathbf{S}$ is irredundant, the minimization operation must be applied on $\mathbf{S}_{\mathsf{pre}}$ if we need an irredundant k-ucCST.

Example 6 *Let us consider the k-ucCST $\mathbf{S}$ of Figure 5.9(a). The computation of the* pre *operation for the transfer $\langle\{p_5, p_6\}, p_2\rangle$ and the bound 1 works as follows. First, the set of minimal elements from which $cones(\mathbf{S})$ is reachable without moving a token from $\{p_5, p_6\}$ to p_2 is computed (Figure 5.9(b)). The application of the transfer on those minimal elements has*

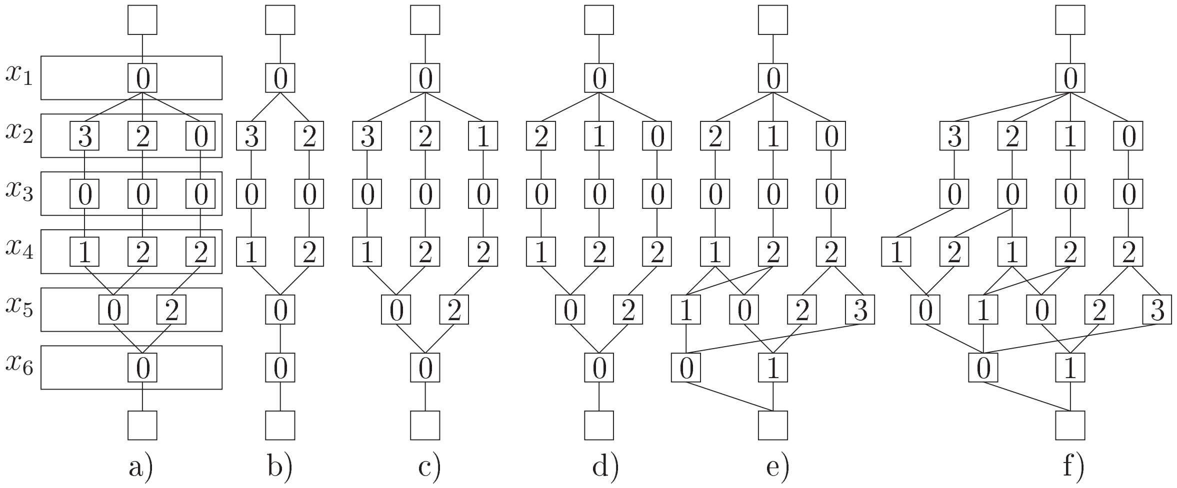

Figure 5.9: k-ucCSTs generated during the computation of pre for the transfer $\langle\{p_5, p_6\}, p_2\rangle$ with the bound 1.

no effect. Then, k-ucCST_Pre$^{\leftarrow}$ *is applied on* $\mathbf{S}$*. First, the elements that are not reachable by moving one tokens from* $\{p_5, p_6\}$ *to* p_2 *are removed (lines 1-3 of Algorithm 16), i.e. here the tuple* $\langle 0, 0, 0, 2, 2, 0\rangle$ *is removed from cones*$(\mathbf{S})$ *(Figure 5.9(c)). Next, one token is removed from the target place* p_2 *(Figure 5.9(d)) and moved to the source places* $\{p_5, p_6\}$ *by applying the algorithm* Add_Tokens *(Figure 5.9(e)) (lines 4-6 of Algorithm 16). Finally, the result of the application of* k-ucCST_Pre$^{\leftarrow}$ *together with the set of minimal markings on which the transfer has no effect is shown in Figure 5.9(f). This last k-ucCST describes the predecessor set of cones*$(\mathbf{S})$ *for the transfer* $\langle\{p_5, p_6\}, p_2\rangle$ *with the bound* 1*.*

In Section 5.5.5, we report experiments using the algorithms presented in this section for the verification of upward closed properties on MTNs corresponding to abstractions of multi-threaded Java programs.

5.3 Forward Approach

Algorithms implementing the forward approach are not guaranteed to terminate, even in the case of monotonic systems. Several techniques have been proposed to force termination of these algorithms in practice. One of these techniques consists in using *acceleration techniques*. An acceleration technique defines a method to compute in a finite number of step an infinite number of reachable states, corresponding often to states reached by iterating a sequence of transitions of the system. If transitions are described through functions, such techniques correspond, given a function f, to construct a relation f^* corresponding to apply any number of time f, i.e. $f^*(s)$ for some state s will return the set $\{s' \mid \exists i : s' = f^i(s)\}$ where f^i corresponds to i sequential applications of f. Acceleration techniques have been first investigated by B. Boigelot and P. Wolper in [BW94] to compute the set of reachable states of automata extended with counters over integer numbers. This technique has been also investigated on automata with counters by A. Finkel and J. Leroux in [FL02], and applied to the verification of other kind of systems like systems communicating through FIFO channels by B. Boigelot and P.

Godefroid (see [BG96]) and systems communicating through lossy FIFO channels by P.A. Abdulla, A. Bouajjani et al. (see [ABJ98]).

According to undecidability results on these models, accelerations do not guarantee the algorithms to terminate in all cases, but experiments show that the set of reachable states can be effectively computed on a large set of examples in practice.

To verify upward closed properties on Petri nets, R.M. Karp and R.E. Miller use a technique similar to accelerations, except that they do not compute the exact set of reachable states but the downward closure of the set of reachable markings. They construct a tree, called *coverability tree*, where the nodes correspond to generators (tuples over $\mathbb{N}\cup\{+\infty\}$) of downward closed sets. More precisely, a node labelled by a generator means that there is an infinite number of markings covered by this generator that are reachable from the initial markings.

Notice that downward closed sets are not closed under the post operation for Petri nets, i.e. given a downward closed set $\mathcal{D}$ of markings and a Petri net transition M, the set of successor markings of $\mathcal{D}$ for M could not be a downward closed set. Indeed, consider the downward closed set $\mathcal{D} = \{\langle p_1, p_2, p_3\rangle \mid p_1 \leq 2 \text{ and } p_2 \leq +\infty \text{ and } p3 \leq 4\}$ and the Petri transition M that adds one token in p_2. If we apply M on $\mathcal{D}$, we obtain the set $\{\langle p_1, p_2, p_3\rangle \mid p_1 \leq 2$ and $1 \leq p_2 \leq +\infty$ and $p3 \leq 4\}$ which is not downward closed. For instance, this set contains the tuple $\langle 2, 2, 4\rangle$ but not the tuple $\langle 2, 0, 4\rangle$. Hence, the post operation used by Karp and Miller on generators does not correspond to the post operation applied on the sets represented by the generators, but corresponds to the downward closure of the set obtained by applying the post operation on the sets represented by generators.

A path from a generator $\mathbf{m}$ to another one $\mathbf{m}'$ means that there is at least one reachable marking covered by $\mathbf{m}'$ that is reachable from a reachable marking covered by $\mathbf{m}$. This tree allows them to detect sequence of Petri net transitions that increase the markings. A path from a generator $\mathbf{m}$ to $\mathbf{m}'$ with $\mathbf{m} \prec \mathbf{m}'$ means that there is a reachable marking covered by $\mathbf{m}$ from which we can reach a greater marking covered by $\mathbf{m}'$ by firing a sequence of Petri net transitions (corresponding to the path from $\mathbf{m}$ to $\mathbf{m}'$). By the monotonicity property of Petri nets, if we continue to apply this sequence of transitions, then the markings will continue to grow. A new generator is then computed to cover all those strictly increasing reachable markings. This generator is constructed from $\mathbf{m}'$ by putting $+\infty$ in the dimensions that growth between $\mathbf{m}$ and $\mathbf{m}'$.

More precisely, the Karp-Miller acceleration function is defined as follows.

Definition 58 (The Karp-Miller Acceleration Function) Let $\mathbf{m}$ and $\mathbf{m}'$ be two (extended) markings with $\mathbf{m} \prec \mathbf{m}'$. $\mathsf{Acceleration}(\mathbf{m}, \mathbf{m}')$ returns the marking $\mathbf{n}$ such that for each place p

- $\mathbf{n}(p) = \mathbf{m}'(p)$ if $\mathbf{m}(p) = \mathbf{m}'(p)$;
- $\mathbf{n}(p) = +\infty$ if $\mathbf{m}(p) < \mathbf{m}'(p)$.

Algorithm 17 is similar to the algorithm of Karp and Miller.

In the following, we will always use the term *ω-marking* instead of generator of downward closed sets of markings. So a ω-marking is a tuples where the value of each component is taken from $\mathbb{N} \cup \{+\infty\}$[5]. We extend the enabling and the firing of a multi-transfer (and hence of a Petri net transition) from markings to ω-markings by considering that $+\infty + c = +\infty$ and $+\infty - c = +\infty$ for any $c \in \mathbb{N}$ and $+\infty + +\infty = +\infty$.

[5]Notice that a marking is also a ω-marking.

Algorithm 17: Forward_Algorithm_For_Petri_Nets

Input: $N = \langle \mathcal{P}, \mathcal{B} \rangle$: a Petri net; S_0 : a set of (downward closed) initial markings ;
S_U : the set of minimal markings of the upward closed set U

Result: *boolean*

begin

$F \leftarrow S_0$;

$fixpoint \leftarrow \emptyset$;

$T \leftarrow \{\langle \top, \mathbf{m} \rangle | \mathbf{m} \in S_0\}$;

1 **while** $\exists \mathbf{m} \in F : \forall \mathbf{m}' \in fixpoint : \mathbf{m} \not\preccurlyeq \mathbf{m}'$ **do**

Let $\mathbf{m}$ such that $\mathbf{m} \in F$ and $\forall \mathbf{m}' \in fixpoint : \mathbf{m} \not\preccurlyeq \mathbf{m}'$;

2 $F \leftarrow F \setminus \{\mathbf{m}\}$;

3 $fixpoint \leftarrow fixpoint \cup \{\mathbf{m}\}$;

foreach $t \in \mathcal{B}$ *with* $\mathbf{m} \rightarrowtail^t \mathbf{n}$ **do**

if $\exists \mathbf{m}' \in fixpoint : \mathsf{Path}(\mathbf{m}', \mathbf{m})$ *and* $\mathbf{m}' \prec \mathbf{n}$ **then**

Let $\mathbf{m}'$ be such that $\mathsf{Path}(\mathbf{m}', \mathbf{m})$ and $\mathbf{m}' \prec \mathbf{n}$;

4 $\mathbf{n} \leftarrow \mathsf{Acceleration}(\mathbf{m}', \mathbf{n})$;

5 $T \leftarrow T \cup \{\langle \mathbf{m}, \mathbf{n} \rangle\}$;

6 $F \leftarrow F \cup \{\mathbf{n}\}$;

if $\exists \mathbf{m} \in S_U, \mathbf{m}' \in fixpoint : \mathbf{m} \preccurlyeq \mathbf{m}'$ **then**

return *true* ;

else

return *false* ;

end

The sets F is the set of ω-markings for which the successors have not been computed yet and $fixpoint$ is the set of already computed ω-markings. The set T is a set of pairs of ω-markings and describes the coverability tree used to compute the acceleration, i.e. a pair $\langle \mathbf{m}, \mathbf{m}' \rangle$ means that there is an edge from $\mathbf{m}$ to $\mathbf{m}'$ in the coverability tree. Finally, the function Path applied on the markings $\mathbf{m}$ and $\mathbf{n}$ returns $true$ if there is a path from $\mathbf{m}$ to $\mathbf{n}$ in the tree described by T. Otherwise, it returns $false$. Path corresponds to search a path between two ω-markings in the tree. Intuitively, while there is at least one ω-marking in F that is not covered by an already treated ω-marking (line 1 of Algorithm 17), the algorithm proceeds as follows. First, it takes a ω-marking $\mathbf{m}$ of F and removes it from F (lines 2-3 of Algorithm 17). Then, it computes the successors of $\mathbf{m}$. For each successor $\mathbf{n}$ of $\mathbf{m}$, if a path from a predecessor ω-marking $\mathbf{m}'$ with $\mathbf{m}' \prec \mathbf{n}$ exists in the tree, then a sequence of transitions that allows the markings to growth is detected and the acceleration function is applied on $\mathbf{m}'$ and $\mathbf{n}$ (line 4 of Algorithm 17). Notice that if there exist several such predecessors, only one is used to compute accelerations. Finally, the new ω-marking is added into the tree (line 5) and F (line 5).

As previously described, Algorithm 17 computes a finite generator set of the downward closure of the set of reachable markings from S_0 (that is not downward closed in general). Let $\mathcal{D}$ be the downward closed set generated by the generator set computed by Algorithm 17 when applied on the Petri net N and the set S_0 of initial markings. The two following important properties holds.

1. Algorithm 17 terminates (see [KM69]);
2. Algorithm 17 allows us to decide the reachability of upward closed sets of markings (see [KM69]) i.e. we have

   $$\mathcal{D} \cap U = \emptyset \text{ if and only if } \mathsf{Reach}_N(S_0) \cap U = \emptyset$$

   for any upward closed set U.

As we will see in the following, a similar approach cannot be extended to the full class of MTN. However, Algorithm 17 can be still applied to a MTN $\mathcal{M}$: if the downward closed set $\mathcal{D}$ has an empty intersection with U, we can still conclude that U is not reachable from the initial markings. Actually, the following holds in this case.

$$\mathcal{D} \cap U = \emptyset \text{ implies that } \mathsf{Reach}_{\mathcal{M}}(S_0) \cap U = \emptyset$$

but

$$\mathcal{D} \cap U \neq \emptyset \text{ does not implies that } \mathsf{Reach}_{\mathcal{M}}(S_0) \cap U \neq \emptyset$$

even when U is an upward closed set.

In this section we show how to apply symbolic techniques on forward algorithms for the verification of safety property on Petri nets and a restricted class of MTNs.

We now define some notions on ω-markings.

Definition 59 (Least Upper Bound) The *least upper bound (lub)* of an infinite set of markings $\{\mathbf{m}_1, \mathbf{m}_2, \ldots\}$ is the ω-marking $\mathbf{m}$ such that:

$$\begin{cases} \mathbf{m}(p) = +\infty & \textbf{if } \forall c \in \mathbb{N} : \exists i \geq 1 : \mathbf{m}_i(p) > c \\ \mathbf{m}(p) = max(\{\mathbf{m}_1(p), \mathbf{m}_2(p), \ldots\}) & \textbf{otherwise}. \end{cases}$$

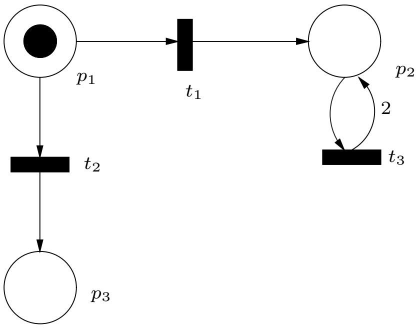

Figure 5.10: A Petri net with the initial marking $\langle 1, 0, 0\rangle$ (where the places are encoded as $\langle p_1, p_2, p_3\rangle$) with the minimal coverability set $\{\langle 1, 0, 0\rangle, \langle 0, +\infty, 0\rangle, \langle 0, 0, 1\rangle\}$.

We are now equipped to define the notion of *coverability set*, which is an important tool to answer safety problems on MTNs.

Definition 60 (Coverability Set of MTN ([Fin90])) A *coverability set* of a MTN $\mathcal{M}$ and a set S_0 of initial markings, noted $\mathtt{CS}(\mathcal{M}, S_0)$, is a set of ω-markings such that :

(1) $\forall \mathbf{m} \in \mathtt{Reach}(\mathcal{M}, S_0)$, there exists $\mathbf{m}' \in \mathtt{CS}(\mathcal{M}, S_0)$ such that $\mathbf{m} \preccurlyeq \mathbf{m}'$.

(2) for every $\mathbf{m} \in \mathtt{CS}(\mathcal{M}, S_0) \setminus \mathtt{Reach}(\mathcal{M}, S_0)$, there exists an infinite sequence $\mathbf{m}_1 \prec \mathbf{m}_2 \prec \mathbf{m}_3 \ldots$ with for all i $\geq 1 : \mathbf{m}_i \in \mathtt{Reach}(\mathcal{M}, S_0)$ and such that $\mathbf{m} = lub(\{\mathbf{m}_1, \mathbf{m}_2, \ldots\})$;

Notice that for a MTN $\mathcal{M}$ and a set S_0 of initial markings, there exist many coverability sets. Hence, in the following, $\mathtt{CS}(\mathcal{M}, S_0)$ will denote one fixed coverability set.

Among all the coverability sets, there is an interesting one, called the minimal coverability set.

Definition 61 (Minimal Coverability Set of MTN ([Fin90])) The *minimal coverability set* of a MTN $\mathcal{M}$ for a set S_0 of initial markings is the intersection of all the finite coverability sets of $\mathcal{M}$ for S_0.

Notice that such a minimal coverability set exists as it corresponds to the minimal finite generator set of the downward closure of the set of markings reachable from the initial ones, which is guaranteed to exist and to be finite following Corollary 6.

Figure 5.10 shows an example of Petri net N with an initial marking where there is one token in the place p_1 and no tokens in the other places. The transition t_1 moves the tokens from p_1 to p_2. When the token is in p_2, then the transition t_3 can be fired as much as we want. Transition t_2 moves the initial token from p_1 to p_3. When the token is in p_3, since there is no transition firable from the marking where there is one token in p_3 and no tokens in the other places, N is blocked. So, if we encode the places as $\langle p_1, p_2, p_3\rangle$, the reachability set of N is the set $\{\langle 1, 0, 0\rangle\} \cup \{\langle 0, n, 0\rangle \mid \text{with } n \geq 1\} \cup \{\langle 0, 0, 1\rangle\}$. A possible (finite) coverability set of N with the initial marking $\langle 1, 0, 0\rangle$ is $\{\langle 1, 0, 0\rangle\} \cup \{\langle 0, 2, 0\rangle\} \cup \{\langle 0, +\infty, 0\rangle\} \cup \{\langle 0, 0, 1\rangle\}$. This coverability set is not the minimal one. Indeed, if we intersect this coverability set with the following one (which is the minimal coverability set) $\{\langle 1, 0, 0\rangle\} \cup \{\langle 0, +\infty, 0\rangle\} \cup \{\langle 0, 0, 1\rangle\}$, the tuple $\langle 0, 2, 0\rangle$ will not appear in the resulting set.

Let us now recall some properties of coverability sets, see [Fin93] for details.

1. any $\texttt{CS}(\mathcal{M}, S_0)$ is an approximation of the reachable markings in the following sense :

$$dcone(\texttt{Reach}(\mathcal{M}, S_0)) \subseteq dcone(\texttt{CS}(\mathcal{M}, S_0))$$

2. any (finite) $\texttt{CS}(\mathcal{M}, S_0)$ is sufficient to answer any safety verification problem if the set of bad markings U is upward closed. In fact, we have in this case:

$$\texttt{Reach}(\mathcal{M}, S_0) \cap U = \emptyset \text{ if and only if } dcone(\texttt{CS}(\mathcal{M}, S_0)) \cap U = \emptyset$$

3. any (finite) $\texttt{CS}(\mathcal{M}, S_0)$ is sufficient to give an answer to the place boundedness problem.

It is shown in [Fin93] that the minimal coverability set is unique and can be computed effectively for any Petri Net N and any downward closed set of initial markings S_0. Algorithm 18 (inspired from the algorithm given in [Fin93]) computes the minimal coverability set of a Petri net and a downward closed set of initial markings S_0. Mainly, this algorithm differs from Algorithm 17 on the following points: (i) an ω-marking that subsumes another one in the tree is not added into the tree (line 1 of Algorithm 18), (ii) when the acceleration function is applied on $\mathbf{m}$ and $\mathbf{n}$, the subtree rooted by $\mathbf{m}$ is replaced by a node labeled by the resulting ω-markings of the acceleration (lines 3-7) (Notice that the acceleration function is applied as much as possible (line 2)), and (iii) when $\mathbf{n}$ is added to the tree, all the subtrees rooted by an ω-marking smaller than $\mathbf{n}$ are removed (line 8-11).

A subtree rooted by a ω-marking $\mathbf{m}$ is removed from T as follows. Remember that T contains all the arcs of the tree computed by the algorithm. First, all the ω-markings reachable from $\mathbf{m}$ are computed and removed from the set of already computed markings $fixpoint$ (line 4 and line 10). Then, all the arcs in the tree from or to a ω-marking that is not in $fixpoint$, i.e. all the arcs from or to the ω-markings that have been removed, are removed from T (line 7 and line 11).

Algorithm 18 greatly improves the practical effectiveness of the forward approach based on Algorithm 17 that computes a coverability set (but not the minimal one in general). This is due to the fact that this algorithm removes useless subtrees (lines 3-11). Indeed, this implies that it manipulates smaller trees and smaller sets than Algorithm 17.

In the next section, we present a symbolic version of Algorithm 18. Then, we show in Section 5.3.2 how to extend the forward approach to a sub-class of MTN.

5.3.1 Symbolic Forward Reachability Algorithm for Petri Nets

A coverability set of a Petri net N with the set of initial markings S_0 is a generator set of the smallest downward closed set that over-approximates the set of markings reachable from S_0 ($dcones(\mathsf{Reach}_N(S_0)) = dcones(\texttt{CS}(N, S_0))$). It follows that Algorithm 18 can be seen as manipulating downward closed sets through their generator set. An improvement of this algorithm is to symbolically handle the sets of ω-markings labeling the nodes of the tree instead of enumerating them, i.e. manipulate sets of markings using k-dcCSTs. Algorithm 19 is such a symbolic forward algorithm that uses k-dcCST to represent the set of markings. The operations k-dcCST_Subsumption, k-dcCST_Union, k-dcCST_NormalForm, k-dcCST_Minimization and k-dcCST_Sat correspond respectively to the operations CST_Subsumption, CST_Union, CST_NormalForm, CST_Minimization and CST_Sat applied on k-dcCSTs.

As mentioned in page 167, downward closed sets of markings are not closed under the post operation for Petri nets. Actually, the post operation on the generator elements used

---

Algorithm 18: Improved_Forward_Algorithm

Input: $N = \langle \mathcal{P}, \mathcal{B} \rangle$: a Petri net; S_0 : a set of (downward closed) initial markings ;
S_U : the set of minimal markings of the upward closed set U
Result: *boolean*
begin
  $F \leftarrow S_0$;
  $fixpoint \leftarrow \emptyset$;
  $T \leftarrow \{\langle \top, \mathbf{m} \rangle | \mathbf{m} \in S_0\}$;
  while $F \neq \emptyset$ **do**
    Let $\mathbf{m}$ such that $\mathbf{m} \in F$;
    foreach $M \in \mathcal{B}$ *such that* M *is firable from* $\mathbf{m}$ **do**
      Let $\mathbf{n}$ be such that $\mathbf{m} \rightarrowtail^{M} \mathbf{n}$;
1   **if** $\nexists \mathbf{m}' \in fixpoint : \mathbf{n} \preccurlyeq \mathbf{m}'$ **then**
        $T \leftarrow T \cup \{\langle \mathbf{m}, \mathbf{n} \rangle\}$;
2   **while** $\exists \mathbf{m}' \in fixpoint : \mathsf{Path}(\mathbf{m}', \mathbf{n})$ *and* $\mathbf{m}' \preccurlyeq \mathbf{n}$ **do**
          Let $\mathbf{m}'$ be such that $\mathsf{Path}(\mathbf{m}', \mathbf{n})$ and $\mathbf{m}' \preccurlyeq \mathbf{n}$;
          $\mathbf{n} \leftarrow \mathsf{Acceleration}(\mathbf{m}', \mathbf{n})$ $(*)$;
3   Let $\langle \mathbf{m}'', \mathbf{m}' \rangle$ be the tuple in T where $\mathbf{m}'$ occurs as second component;
4   Replace $\langle \mathbf{m}'', \mathbf{m}' \rangle$ by $\langle \mathbf{m}'', \mathbf{n} \rangle$ in T ;
5           $fixpoint \leftarrow fixpoint \setminus \{\mathbf{m}'' | \mathsf{Path}(\mathbf{m}', \mathbf{m}'')\}$;
6           $F \leftarrow F \setminus \{\mathbf{m}'' | \mathsf{Path}(\mathbf{m}', \mathbf{m}'')\}$;
7           $T \leftarrow T \setminus \{\langle \mathbf{m}''', \mathbf{m}'' \rangle | \mathbf{m}'' \notin fixpoint\}$;
8   **foreach** $\mathbf{m}'' \in fixpoint : \mathbf{m}'' \preccurlyeq \mathbf{n}$ **do**
9           $fixpoint \leftarrow fixpoint \setminus \{\mathbf{n}' | \mathsf{Path}(\mathbf{m}'', \mathbf{n}')\}$;
10           $F \leftarrow F \setminus \{\mathbf{n}' | \mathsf{Path}(\mathbf{m}'', \mathbf{n}')\}$;
11           $T \leftarrow T \setminus \{\langle \mathbf{n}', \mathbf{n}'' \rangle | \mathbf{n}' \notin fixpoint$ or $\mathbf{n}'' \notin fixpoint\}$;
        $fixpoint \leftarrow fixpoint \cup \{\mathbf{n}\}$;
        $F \leftarrow F \cup \{\mathbf{n}\}$;
  if $\exists \mathbf{m} \in S_U, \mathbf{m}' \in fixpoint : \mathbf{m} \preccurlyeq \mathbf{m}'$ **then**
    return *true* ;
  else
    return *false* ;
end

---

by forward algorithms corresponds to compute the downward closure of the post operation, written $\mathsf{post}^{\downarrow}$, on the downward closed set that they represent.

Definition 62 (The $\mathsf{post}^{\downarrow}$ operation) Let S be a downward closed set of markings and let M be a Petri net transition. We define $\mathsf{post}^{\downarrow}$ for S and M as

$$\mathsf{post}^{\downarrow}(S, M) = \{\mathbf{m} \mid \exists \mathbf{m}' \in S : \mathbf{m}' \rightarrowtail^{M} \mathbf{m}'' \text{ and } \mathbf{m} \preccurlyeq \mathbf{m}''\}$$

The operation $\mathsf{k\text{-}dcCST\_Post}^{\downarrow}$ used by Algorithm 19 corresponds to the operation $\mathsf{post}^{\downarrow}$ applied on k-dcCSTs. For a downward closed set S and its generator set $gen(S)$, the following holds for any Petri net transition M.

$$\mathsf{post}^{\downarrow}(S, M) = dcone(\mathsf{post}(gen(S), M))$$

Indeed, by monotonicity, the ω-markings reached from the generator elements of S by applying M cover all the markings reachable from the markings of S.

To represent the tree, i.e. the set of arcs between ω-markings used to compute the acceleration function, k-sharing trees are used. In [Fin93], the tree is constructed to (i) detect when the acceleration function must be applied (line 2 of Algorithm 18) and (ii) remove useless ω-markings (line 8-11 of Algorithm 18). But for those cases we only need the reachability relation between ω-markings rather than the direct successor relation of the tree. For this reason, we do not compute a coverability tree in the symbolic algorithm: we do not maintain the successor relation but the transitive closure of that relation instead. It is easy to see that this is sufficient to answer upward closed set reachability problems or place boundedness problems.

More precisely, Algorithm 19 works as follows. The ω-markings that must be still treated are stored into the k-dcCST $\mathbf{F}$, the set of ω-markings already treated are stored into the k-dcCST $\mathbf{R}$, and the transitive closure of the accessibility relation (the tree constructed by Algorithm 18) is stored into the k-sharing tree T^{+}. Lines 1-2 of Algorithm 19 correspond to line 1 of Algorithm 18 and remove from the set of newly computed ω-markings those ones that are covered by an already computed ω-marking. Lines 3-4 consist to add into the transitive closure of the tree the newly computed ω-markings. Line 5 of Algorithm 19 consists in applying the acceleration function of Karp and Miller and corresponds to lines 2-8 of Algorithm 18. Finally, line 6-10 of Algorithm 19 correspond to line 9-12 of Algorithm 18 and consist to remove non-maximal ω-markings and their successors (the useless subtrees) from the set of already managed ω-markings ($\mathbf{R}$). Unlike in the enumerative version of the algorithm, no ω-markings are removed from the frontier, i.e. the set of ω-markings that have not been managed yet, when removing the useless sub-trees. This is due to the following problem.

Suppose that the ω-markings of the frontier were removed when removing the useless subtrees. Suppose that, at some iteration i of the *while* loop, Algorithm 19 computes two ω-markings $\mathbf{m}_1$ and $\mathbf{m}_2$ that have the following properties. $\mathbf{m}_1$ has a predecessor $\mathbf{m}'_1$ and $\mathbf{m}_2$ has a predecessor $\mathbf{m}'_2$, i.e. we have $\{\langle \mathbf{m}'_1, \mathbf{m}_1\rangle, \langle \mathbf{m}'_2, \mathbf{m}_2\rangle\} \subseteq elem(T^{+})$ with $\{\mathbf{m}_1, \mathbf{m}_2\} \subseteq elem(\mathbf{F})$, such that $\mathbf{m}'_1 \prec \mathbf{m}_2$ and $\mathbf{m}'_2 \prec \mathbf{m}_1$. Following the principles of Algorithm 18 (lines 9-12) extended to sets of ω-markings, Algorithm 19 would remove the useless subtrees rooted by the nodes $\mathbf{m}'_1$ and $\mathbf{m}'_2$ leading to remove both $\mathbf{m}_1$ **and** $\mathbf{m}_2$. However, in the enumerative algorithm only one subtree rooted either by $\mathbf{m}_1$ or by $\mathbf{m}_2$ is removed and the other one is developed since the ω-markings of the frontier are treated enumeratively. We conclude that the symbolic version of the algorithm would not compute a coverability tree if the ω-markings of the frontier are removed when removing the useless subtrees.

Algorithm 19: Symbolic_Forward_Algorithm

Input: $N = \langle \mathcal{P}, \mathcal{B} \rangle$: a Petri net; $\mathbf{S}_0$: a k-dcCST; $\mathbf{m}_U$: a marking
Result: a boolean
begin
  $\mathbf{F} \leftarrow \mathsf{Copy}_{\text{k-dcCST}}(\mathbf{S}_0)$;
  $\mathbf{R} \leftarrow$ k-dcCST_Empty ;
  while not(k-dcCST_Subsumption($\mathbf{F}, \mathbf{R}$)) **do**
    $\mathbf{T} \leftarrow$ k-dcCST_Empty_set;
    foreach $M \in \mathcal{B}$ **do**
      $\mathbf{N} \leftarrow$ k-dcCST_Post$^{\downarrow}(\mathbf{F}, M)$;
      $\mathbf{T} \leftarrow$ k-dcCST_Union($\mathbf{T}, \mathbf{N}$);
1    $\mathbf{T} \leftarrow$ k-dcCST_NormalForm($\mathbf{T}$) ;
2    $\mathbf{T} \leftarrow$ k-dcCST_Minimization($\mathbf{T}, \mathbf{R}$) ;
3    $T^+ \leftarrow \mathsf{union}(T^+, \{\langle \mathbf{m}, \mathbf{m}' \rangle | \mathbf{m} \in elem(\mathbf{F}) \wedge \mathbf{m}' \in elem(\mathbf{T}) \wedge \exists M \in \mathcal{B} : \mathbf{m} \rightarrowtail^M \mathbf{m}'\})$;
4    $T^+ \leftarrow \mathsf{union}(T^+, \{\langle \mathbf{m}, \mathbf{m}' \rangle | \mathbf{m}' \in elem(\mathbf{T}) \wedge \exists \mathbf{m}'' \in elem(\mathbf{F}), M \in \mathcal{B} : \mathbf{m}'' \rightarrowtail^M \mathbf{m}' \wedge \langle \mathbf{m}, \mathbf{m}'' \rangle \in T^+\})$;
5    $\mathbf{newF}, T^+ \leftarrow Acc(\mathbf{R}, \mathbf{T}, T^+)$;
    $\mathbf{newF} \leftarrow$ k-dcCST_NormalForm($\mathbf{newF}$) ;
6    $\mathbf{R} \leftarrow$ k-dcCST_Minimization($\mathbf{R}, \mathbf{newF}$) ;
7    $useless \leftarrow$ the k-dcCST $\mathbf{S}$ such that $elem(\mathbf{S}) = \{\mathbf{m} | \exists \langle \mathbf{m}', \mathbf{m} \rangle \in T^+ : \mathbf{m}' \notin \mathbf{R}\}$;
8    $\mathbf{R} \leftarrow \mathsf{minus}(\mathbf{R}, useless)$;
9    $\mathbf{R} \leftarrow$ k-dcCST_Union($\mathbf{R}, \mathbf{F}$);
    $\mathbf{F} = \mathbf{newF}$;
    $I \leftarrow$ Cartesian_Product(k-dcCST_Union($\mathbf{R}, \mathbf{F}$), k-dcCST_Union($\mathbf{R}, \mathbf{F}$)) ;
10    $T^+ \leftarrow \mathsf{intersection}(T^+, I)$;
  if k-dcCST_Sat($\mathbf{m}_U, \mathbf{R}$) **then**
    return true ;
  else
    return false ;
end

Algorithm 20: Acc

Input: S : k-dcCST that represents the set of treated markings; $Succ$: k-dcCST that represents the set of markings that remain to treat; T^+ : k-sharing tree that represents the transitive closure of the tree

Result: (i) a k-dcCST that represents the (new) set of ω-markings to treat after the acceleration, (ii) and a k-sharing tree that represents the transitive closure of the tree after the acceleration

begin

$G \leftarrow$ the k-dcCST $\mathbf{S}$ such that $elem(\mathbf{S}) = \{\langle \mathbf{m}, \mathbf{m}' \rangle \in T^+ : \mathbf{m} \prec \mathbf{m}'\}$;

$Accelerated \leftarrow$ the k-dcCST $\mathbf{S}$ such that $elem(\mathbf{S}) = \{\mathbf{m} \mid \exists \langle \mathbf{m}', \mathbf{m}'' \rangle \in G : \mathbf{m} = f^a(\mathbf{m}', \mathbf{m}'')\}$;

$H \leftarrow \{\langle \mathbf{m}, \mathbf{m}' \rangle | \exists \langle \mathbf{m}''', \mathbf{m}'' \rangle \in G : \mathbf{m}' = f^a(\mathbf{m}''', \mathbf{m}'') \wedge \langle \mathbf{m}, \mathbf{m}''' \rangle \in T^+\}$;

$T^+ \leftarrow \mathsf{union}(T^+, H)$;

$Succ \leftarrow \mathsf{k\text{-}dcCST\_Union}(Succ, Accelerated)$;

$Useless \leftarrow$ the k-dcCST $\mathbf{S}$ such that $elem(\mathbf{S}) = \{\mathbf{m} | \exists \mathbf{m}' : \langle \mathbf{m}, \mathbf{m}' \rangle \in G\}$;

$tmp \leftarrow \{\mathbf{m} | \exists \mathbf{m}' \in Useless : \langle \mathbf{m}', \mathbf{m} \rangle \in T^+\}$;

$Useless \leftarrow \mathsf{k\text{-}dcCST\_Union}(tmp, Useless)$;

$T^+ \leftarrow \mathsf{minus}(T^+, \{\langle \mathbf{m}, \mathbf{m}' \rangle | \mathbf{m} \in Useless\})$;

$T^+ \leftarrow \mathsf{minus}(T^+, \{\langle \mathbf{m}, \mathbf{m}' \rangle | \mathbf{m}' \in Useless\})$;

$Succ \leftarrow \mathsf{minus}(Succ, Useless)$;

return $Succ, T^+$;

end

In Algorithm 19, T^+ is represented with k-sharing threes where the domain of values is $\mathbb{N} \cup \{+\infty, \top, \bot\}$. Some relational operations presented in [Zam97] are used to manipulate T^+. These operations are the selection, defined as

$$\mathsf{Select}(\mathbf{S}, i, j) = \mathbf{T} \text{ such that } elem(\mathbf{T}) = \{\mathbf{m} \in elem(\mathbf{S}) | m_j = m_i\}$$

the projection, defined as

$$\begin{aligned}\mathsf{Projection}(\mathbf{S}, i) = \mathbf{T} \text{ such that } elem(\mathbf{T}) = \{&\langle m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_n \rangle | \\ &\exists \langle m_1, \ldots, m_{i-1}, m_i, m_{i+1}, \ldots, m_n \rangle \in elem(\mathbf{S})\}\end{aligned}$$

and the Cartesian product, defined as

$$\begin{aligned}\mathsf{Cartesian\_Product}(\mathbf{S}, \mathbf{T}) = \mathbf{I} \text{ such that } elem(\mathbf{I}) = \{&\langle m_1, m_1', \ldots, m_n, m_n' \rangle | \\ &\langle m_1, \ldots, m_n \rangle \in elem(\mathbf{S}) \text{ and } \langle m_1', \ldots, m_n' \rangle \in elem(\mathbf{T})\}\end{aligned}$$

The definition of Cartesian_Product given here differs from the definition given in [Zam97] in the following way. In [Zam97], a sequential scheme is used, i.e. the Cartesian product between the k-sharing tree $\mathbf{S}_1$ with $elem(\mathbf{S}_1) = \{\langle m_1, \ldots, m_n \rangle\}$ and the k-sharing tree $\mathbf{S}_2$ with $elem(\mathbf{S}_2) = \{\langle m_1', \ldots, m_n' \rangle\}$ returns a k-sharing tree $\mathbf{S}$ with $elem(\mathbf{S}) = \{\langle m_1, \ldots, m_n, m_1', \ldots, m_n' \rangle\}$.

Our definition uses an interleaving encoding, i.e. $\mathbf{S} = \mathsf{Cartesian\_Product}(\mathbf{S}_1, \mathbf{S}_2)$ is such that $elem(\mathbf{S}) = \langle m_1, m_1', m_2, m_2', \ldots, m_n, m_n' \rangle$. This is an important property that allows us to efficiently compute the acceleration function. Indeed, to apply the acceleration function, tuples $\langle \mathbf{m}, \mathbf{m}' \rangle$ with $\mathbf{m} \prec \mathbf{m}'$ must be detected. To detect such pairs in T^+ using a interleaving encoding, it suffices to compare the value of nodes encoding the first component of pairs with the value of their successor nodes, that encode the second component of pairs. In other words, as the two components of pairs are interleaved, their comparison can be done with local tests on nodes and their successors. Algorithm 31 shown in Appendix (that uses the recursive procedures described by Algorithm 29 and Algorithm 30 that are also shown in Appendix) computes the Cartesian product on k-sharing trees with an interleaving encoding.

In the following, we will sometimes compute the Cartesian product between a set S_1 of tuple of the form $\langle \mathbf{m}_1, \mathbf{m}_2 \rangle$ and a set S_2 of ω-markings encoded through k-sharing trees. As our Cartesian product only works on sets of tuples of the same size, we need a special encoding of the set S_2. To compute the Cartesian product by using Algorithm 31, we first compute $S_3 = \mathsf{Cartesian\_Product}(S_2, S_2)$. Then we compute $\mathsf{Cartesian\_Product}(S_1, S_3)$. This leads to a set of tuples $\langle \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3, \mathbf{m}_4 \rangle$ where $\langle \mathbf{m}_1, \mathbf{m}_3 \rangle$ is a tuples of S_1, $\mathbf{m}_2 = \mathbf{m}_4$ and $\mathbf{m}_2 \in S_2$. In the following, we will omit these details and we will assume that we can construct the Cartesian product between a set of tuple of ω-markings and a set of ω-markings.

The various operations on T^+ are computed as follows. The operation of line 3 of Algorithm 19 consists in adding the direct successor relation between the ω-markings of the frontier (in $elem(\mathbf{F})$) and the ω-markings computed with the $\mathsf{k\text{-}dcCST\_Post}^{\downarrow}$ operation. This operation is computed as follows. First, we compute the set of tuples $\langle \mathbf{m}, \mathbf{m} \rangle$ with $\mathbf{m} \in elem(\mathbf{F})$. This is achieved by constructing $S = \mathsf{Cartesian\_Product}(\mathbf{F}, \mathbf{F})$. Hence we have that $elem(S)$ corresponds to the set of pairs $\{\langle \mathbf{m}_1, \mathbf{m}_2 \rangle \mid \mathbf{m}_1 \in \mathbf{F} \text{ and } \mathbf{m}_2 \in \mathbf{F}\}$. Second, only the elements in $elem(S)$ corresponding to pairs of the form $\langle \mathbf{m}, \mathbf{m} \rangle$ are kept by using the selection operation of [Zam97]. Then, the post operation for each transition M is computed on the second component of pairs corresponding to the elements in $elem(S)$. Let S_{post} be the resulting k-sharing tree, i.e. $elem(S_{\mathsf{post}})$ corresponds to the set of pairs of ω-markings $P = \{\langle \mathbf{m}_1, \mathbf{m}_2\rangle \mid \mathbf{m}_1 \in \mathbf{F}$ and $\mathbf{m}_1 \rightarrowtail^M \mathbf{m}_2$ for some $M \in \mathcal{B}\}$. Finally, we only want to keep the pairs in P where the second component is an element in $\mathbf{T}$ (the maximal ω-markings computed by $\mathsf{k\text{-}dcCST\_Post}^{\downarrow}$). To achieve this goal, we first compute $\mathsf{Cartesian\_Product}(S_{\mathsf{post}}, \mathbf{T})$. This gives us a k-sharing tree T such that $elem(T)$ corresponds to the set of tuples $P' = \{\langle \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3\rangle \mid \langle \mathbf{m}_1, \mathbf{m}_2\rangle \in P$ and $\mathbf{m}_3 \in elem(\mathbf{T})\}$. Then, by using the selection operation Select we only keep the tuples in $elem(T)$ that correspond to a tuple $\langle \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3\rangle$ where $\mathbf{m}_2$ and $\mathbf{m}_3$ coincide. This leads to the construction of the k-sharing tree T' such that $elem(T')$ corresponds to the set $\{\langle \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3\rangle \in P' \mid \mathbf{m}_2 = \mathbf{m}_3\}$. Finally, the k-sharing tree T'' is constructed by applying the projection operation on T' to remove the third component of the tuples corresponding to elements in $elem(T')$. Hence, $elem(T'')$ corresponds to the set of pairs $\{\langle \mathbf{m}_1, \mathbf{m}_2\rangle \mid \mathbf{m}_1 \in \mathbf{F}, \mathbf{m}_2 \in \mathbf{T}$ and there exists $M \in \mathcal{B}$ such that $\mathbf{m}_1 \rightarrowtail^M \mathbf{m}_2\}$. Finally, the union operation is applied on T'' and $\mathbf{S}_{T^+}$.

The operation of line 4 adds the arcs corresponding to the accessibility relation between the ω-markings computed by $\mathsf{k\text{-}dcCST\_Post}^{\downarrow}$ and all their predecessors into $elem(T^+)$. This corresponds to compute the transitive closure of the reachability relation between the ω-markings computed by $\mathsf{k\text{-}dcCST\_Post}^{\downarrow}$ and all their predecessors. This operation is computed as follows. We first compute $T = \mathsf{Cartesian\_Product}(T^+, T^+)$. Hence, $elem(T)$ corresponds to the set $P = \{\langle \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3, \mathbf{m}_4\rangle \mid \langle \mathbf{m}_1, \mathbf{m}_3\rangle$ and $\langle \mathbf{m}_2, \mathbf{m}_4\rangle$ correspond to an element in $elem(T^+)\}$. The Select operation is then applied on T to only keep in P the tuples of the form $\langle \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3, \mathbf{m}_4\rangle$ where $\mathbf{m}_2 = \mathbf{m}_3$. This leads to the construction of the k-sharing tree T' such that $elem(T')$ corresponds to the set of tuples of ω-markings $\{\langle \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3, \mathbf{m}_4\rangle \mid \mathbf{m}_2 = \mathbf{m}_3, \langle \mathbf{m}_1, \mathbf{m}_3\rangle$ and $\langle \mathbf{m}_2, \mathbf{m}_4\rangle$ correspond to an element in $elem(T^+)\}$. Finally, the $\mathsf{Projection}$ operation is applied on T' to remove the second and the third component of each tuple. We obtain a k-sharing tree corresponding to the set of pairs $\{\langle \mathbf{m}_1, \mathbf{m}_2\rangle \mid \mathbf{m}_2$ is reachable from $\mathbf{m}_1$ in the tree $\}$. Finally, the union operation is applied on that k-sharing tree and T^+.

Intuitively, the line 7 consists in computing all the ω-markings contained by sub-trees rooted by ω-markings that are smaller than one another ω-marking in the tree. Actually, lines 7 and 8 correspond to lines 8-11 of Algorithm 18. The operation is computed as follows. Notice that the ω-markings that are smaller than another ω-marking in $\mathbf{R}$ have already been removed from $elem(\mathbf{R})$ by line 6.

First, the k-sharing tree $T = \mathsf{Cartesian\_Product}(\mathbf{R}, T^+)$ is computed. T is such that $elem(T)$ corresponds to the set of tuples $P = \{\langle \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3\rangle \mid \mathbf{m}_1 \in elem(\mathbf{R})$ and $\langle \mathbf{m}_2, \mathbf{m}_3\rangle$ corresponds to an element in $elem(T^+)\}$. We then use the selection operation to keep the elements where the first component and the second one coincide. This leads to the construction of the k-sharing tree T' such that $elem(T')$ corresponds to the set P' obtained from P by removing the elements $\langle \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3\rangle$ with $\mathbf{m}_1 \neq \mathbf{m}_2$. The k-sharing tree T'' is then constructed by applying the $\mathsf{Projection}$ operation on T' such that $elem(T'')$ corresponds to the set of pairs $\{\langle \mathbf{m}_2, \mathbf{m}_3\rangle \mid \exists \mathbf{m}_1 : \langle \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3\rangle \in P'\}$. That set is exactly the set of pairs corresponding to the elements in $elem(T^+)$ where the first component is an element in $elem(\mathbf{R})$. To obtain the *useless* elements, we compute $T''' = \mathsf{minus}(T^+, T'')$. So, T''' is such that $elem(T''')$ is the set of pairs corresponding to the elements in $elem(T^+)$ where the first component is not an element in $elem(\mathbf{R})$, i.e. where the second component corresponds to an useless ω-marking. Let P'' be the set of pairs corresponding to $elem(T''')$. The set of useless ω-markings is the set $\{\mathbf{m}_2 \mid \exists \mathbf{m}_1 : \langle \mathbf{m}_1, \mathbf{m}_2\rangle \in P''\}$. To obtain the k-sharing tree corresponding to that set, the $\mathsf{Projection}$ operation is applied on T'''.

The treatment related to the acceleration function is given by Algorithm 20. This algorithm can be divided in two distinct parts. The first part (lines 1-5) consists in computing the acceleration itself. In line 1, the tuples $\langle \mathbf{m}, \mathbf{m}' \rangle$ of $elem(T^+)$ such that $\mathbf{m} \prec \mathbf{m}'$ are computed. That pairs correspond to all the paths in the tree from ω-markings $\mathbf{m}$ to $\mathbf{m}'$ with $\mathbf{m} \prec \mathbf{m}'$. This is computed on k-sharing trees by looking at nodes n of odd layers of T^+ and removing the sons n' of n that have a greater value i.e. $val(n) > val(n')$. The algorithm 28 is applied as usual on the resulting pre-sharing tree to ensure condition 6 of Definition 42.

Line 2 makes the acceleration itself using the function f_a that computes the acceleration defined by Definition 58. The algorithm consists in looking at the nodes n of the odd layers of G and replace the value of the sons n' by $+\infty$ when $val(n) < val(n')$. As the resulting pre-sharing tree is not guarantee to ensure conditions 5-6 of Definition 42, Algorithm 27 and Algorithm 28 are applied. Finally, the Projection operation is used to construct the set of accelerated ω-markings.

Line 3 computes the arcs between the new (accelerated) ω-markings and their predecessors. The operation is computed as follows. First, $T = \mathsf{Cartesian\_Product}(T^+, G)$ is computed. Hence, $elem(T)$ corresponds to the set $P = \{\langle \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3, \mathbf{m}_4 \rangle \mid \langle \mathbf{m}_1, \mathbf{m}_3 \rangle$ corresponds to an element in $elem(T^+)$ and $\langle \mathbf{m}_2, \mathbf{m}_4 \rangle$ corresponds to an element in $elem(G)\}$. Then, the Select operation is applied on T to keep only the tuples in P of the form $\langle \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3, \mathbf{m}_4 \rangle$ where $\mathbf{m}_2 = \mathbf{m}_3$. This leads to the construction of the k-sharing tree T' such that $elem(T')$ corresponds to the set $P' = \{\langle \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3, \mathbf{m}_4 \rangle \in P \mid \mathbf{m}_2 = \mathbf{m}_3\}$. Finally, the k-sharing tree H is obtained by applying the Projection operation on T' to keep only the first and the fourth components of each tuple in P'. Hence, we have that $elem(H)$ corresponds to the set $\{\langle \mathbf{m}_1, \mathbf{m}_4 \rangle \mid \exists \mathbf{m}_2, \mathbf{m}_3 : \langle \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3, \mathbf{m}_4 \rangle \in P'\}$.

Those pairs are then added into $elem(T^+)$ (line 4), and the new ω-markings are added into the frontier (line 5).

The second part of the algorithm consists in removing the useless subtree (lines 6-11) rooted by the ω-markings that are in the first component of a tuple $\langle \mathbf{m}_1, \mathbf{m}_2 \rangle$ in $elem(G)$ (the non-maximal elements of R). Mainly, operations at lines 6-10 are computed on k-sharing trees in the same way than the operation of line 3.

First, the set of useless nodes, i.e. the ω-markings in the useless subtrees, is computed (lines 6-8). Then, the set of useless arcs is computed and removed from T^+ (lines 9-10). The treatment terminates by removing the useless ω-markings from the frontier (line 11).

In Section 5.5.4 and Section 5.5.5, Algorithm 19 is evaluated on some examples of Petri nets.

5.3.2 Forward Reachability Algorithm for Extended Petri Nets

Well known results allow us to easily prove that the construction of a coverability set for some extensions of Petri nets is not possible in general. We have a similar result for MTNs. In [DFS98], it is proved that the place-boundedness problem is undecidable for PN + T (MTNs with only unbounded transfers) and we proved in Chapter 2 that the place-boundedness is also undecidable for PN + NBA (MTNs with only transfers bounded to 1). As a finite coverability set allows us to decide such a property, we directly deduce the following theorem.

Theorem 27 There is no algorithm that takes a PN + NBA or a PN + T $\mathcal{M}$ with an initial marking $\mathbf{m}_0$ and returns a finite coverability set of $\mathcal{M}$ for $\mathbf{m}_0$.

From this result, we obtain the following general result on MTNs.

Corollary 13 There is no algorithm that takes a MTN $\mathcal{M}$ with an initial marking $\mathbf{m}_0$ and returns a finite coverability set of $\mathcal{M}$ for $\mathbf{m}_0$.

In this section, we identify however a subclass of MTNs for which we can compute a weaker notion of coverability set that is sufficient to decide the coverability problem, i.e. the reachability of an upward closed set of markings. This class is of practical importance since it covers all the examples of JAVA program abstractions analyzed with the backward approach in [DRB02].

Multi Isolated Transfer Nets

In the following, we will only consider multi-transfers with transfer blocks having the bound $+\infty$ (in that case, firing a multi-transfer is deterministic). Let us define some notations that will be used in this section

Given a multi-transfer $M = \langle T, \{\langle\{B_1, B_2, \ldots, B_u\}, +\infty\rangle\}\rangle$, we use the notation $\mathsf{Transfer}(M)$ to refer to its set of transfers $\{B_1, B_2, \ldots, B_u\}$. Remember that each B_i is a pair $\langle P_i, p_i\rangle$, where P_i is the set of source places and p_i is the target place of the transfer B_i. We extend $\mathsf{Transfer}$ to sets and sequences of multi-transfers in the obvious way.

Given a sequence of multi-transfers σ, a marking $\mathbf{m}$ and a place p, $\mathsf{Sources}(\sigma, p)$ denotes the set $\{p' \mid \exists\langle P, p_t\rangle \in \mathsf{Transfer}(\sigma) : p_t = p \text{ and } p' \in P\}$, and $\mathsf{Effect}(\sigma, \mathbf{m}, p)$ is defined such that $\mathsf{Effect}(\sigma, \mathbf{m}, p) = k$ if σ is firable from $\mathbf{m}$ and $\mathbf{m} \rightarrowtail^{\sigma} \mathbf{m}'$ with $k = \mathbf{m}'(p) - \mathbf{m}(p)$; otherwise, if σ is not firable from $\mathbf{m}$, $\mathsf{Effect}(\sigma, \mathbf{m}, p)$ is undefined.

We define a sub-class of MTN, called *Multi Isolated Transfer Nets*, as follows.

Definition 63 (Multi Isolated Transfer Net) A *Multi Isolated Transfer Net* (MITN for short) $\mathcal{M} = \langle \mathcal{P}, \mathcal{B}\rangle$ is a MTN where all the multi-transfers $M \in \mathcal{B}$ are of the form $M = \langle T, \{\langle\{B_1, B_2, \ldots, B_u\}, +\infty\rangle\}\rangle$. Moreover, $\mathcal{M}$ satisfies the following additional conditions, expressed informally as:

(i) a place cannot be a source of two different transfers with a different target;

(ii) a place cannot be the target of one transfer and a source of another one;

(iii) a place that is a source of a transfer cannot be a source of the Petri Net part of a multi-transfer.

and formally as follows:

(i) $\neg\exists M_i, M_j \in \mathcal{B}$ with $\langle P_i, p_i\rangle \in \mathsf{Transfer}(M_i)$, $\langle P_j, p_j\rangle \in \mathsf{Transfer}(M_j)$ such that $p_i \neq p_j$ and $P_i \cap P_j \neq \emptyset$;

(ii) $\neg\exists M_i, M_j \in \mathcal{B}$ with $\langle P_i, p_i\rangle \in \mathsf{Transfer}(M_i), \langle P_j, p_j\rangle \in \mathsf{Transfer}(M_j)$ with $p_i \in P_j$;

(iii) $\forall M_i \in \mathcal{B}, \forall\langle P_i, p_i\rangle \in \mathsf{Transfer}(M_i), \forall M_j \in \mathcal{B}$ with $M_j = \langle\langle\mathcal{I}, \mathcal{O}\rangle, B\rangle : \mathcal{I}(P_i) = 0$.

Such a MITN is illustrated in Figure 5.11. Notice that Petri nets are MITN. That sub-class is called multi isolated transfer net because the restrictions isolate the source places of transfers in the sense that when a token is put into a source place of a transfer, either it stays into that place or it is transferred to a unique target place (conditions (i) and (iii); condition (ii) is of a more technical nature and its interest will occur later). So, those tokens cannot move anywhere in the net.

We now define a weak notion of coverability set that is parameterized by a (non-empty) upward closed set U. In the rest of this section, we only consider upward closed sets U

with one minimal element $\mathbf{m}_U$. This restriction aims at simplifying the presentation and the extension to a finite set of minimal markings (and so to any upward closed set of markings) is not difficult [6]. In the following, we always denote by $\mathbf{m}_U$ the minimal marking of the upward closed set U.

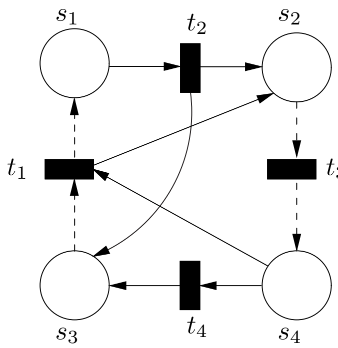

Figure 5.11: An example of MITN.

First, we define the notion of *weak limit* that defines the type of ω-markings (that are not reachable markings) that are contained into a weak coverability set.

Definition 64 (Weak Limit) Given an upward closed set U, and an infinite sequence of markings $\mathcal{S} = \mathbf{m}_1\mathbf{m}_2\ldots$ with $\mathbf{m}_i \prec \mathbf{m}_{i+1}$ for all $i \geq 1$, the ω-marking $\mathbf{m}$ is a *weak limit* according to U (U-weak limit for short) of $\mathcal{S}$ if it satisfies the following conditions.

- if $\mathbf{m}(p) = \omega$, then there exists $i \geq 1$ such that for all $i' \geq i$ we have $\mathbf{m}_{i'}(p) \geq \mathbf{m}_U(p)$;
- if $\mathbf{m}(p) \neq \omega$, then there exists $i \geq 1$ such that for all $i' \geq i$ we have $\mathbf{m}_{i'}(p) = \mathbf{m}(p)$.

Notice that if $\mathbf{m}$ is a U-weak limit of S and $+\infty \neq \mathbf{m}(p) \geq \mathbf{m}_U(p)$ for some place p, replacing $\mathbf{m}(p)$ by $+\infty$ also yields a U-weak limit of S.

Classically, a coverability set is a generator set of the smallest downward closed set of markings that includes the reachability set of a (extended) Petri net. Intuitively, a weak coverability set is a generator set of a downward closed set of markings, but not necessarily the smallest one, that includes the reachability set of a (extended) Petri net and that allows us to decide nevertheless the coverability problem.

Definition 65 (Weak Coverability Set) Given a MTN $\mathcal{M}$ with the set of initial markings S_0 and an upward closed set U generated by the marking $\mathbf{m}_U$, a weak coverability set of $\mathcal{M}$ according to S_0 and U (noted $\mathtt{WCS}(\mathcal{M}, S_0, U)$) is a set of ω-markings such that the following two conditions hold:

(1) $\forall \mathbf{m} \in \mathsf{Reach}_{\mathcal{M}}(S_0)$, there exists $\mathbf{m}' \in \mathtt{WCS}(\mathcal{M}, S_0, U)$ with $\mathbf{m} \preccurlyeq \mathbf{m}'$;

(2) $\forall \mathbf{m} \in \mathtt{WCS}(\mathcal{M}, S_0, U) \setminus \mathsf{Reach}_{\mathcal{M}}(S_0)$, there exists an infinite sequence of markings $\mathcal{S} = \mathbf{m}_1\mathbf{m}_2\ldots\mathbf{m}_n\ldots$ with $\mathbf{m}_i \in \mathsf{Reach}_{\mathcal{M}}(S_0)$ and $\mathbf{m}_i \prec \mathbf{m}_{i+1}$ for any $i \geq 1$ such that $\mathbf{m}$ is a U-weak limit of $\mathcal{S}$.

[6]To test the accessibility of an upward closed set U with the set of minimal markings $\{\mathbf{m}_1, \ldots, \mathbf{m}_n\}$, we test the reachability of the n upward closed sets U_i $(1 \leq i \leq n)$ generated by the minimal marking $\mathbf{m}_i$. If one of those upward closed sets is reachable from the set of initial markings, so does U.

As in the case of coverability sets, there exist many weak coverability sets. In particular, notice that a coverability set is also a weak coverability set. Hence, in the following, $\mathtt{WCS}(\mathcal{M}, S_0, U)$ will denote one fixed weak coverability set of the MTN $\mathcal{M}$ with the set of initial markings S_0 and the upward closed set U.

The downward closure of a weak coverability set over-approximates the set of reachable states (by Definition 65(1)) and the downward closure of the coverability sets since the elements of a weak coverability set are greater or equal to the limit of an infinite strictly increasing sequence of reachable markings or are reachable markings (by Definition 65(2)). Nevertheless, it is still sufficiently precise to verify upward closed safety properties. This is formally expressed by the following proposition:

Proposition 20 Let $\mathcal{M}$ be a MTN with the initial set of markings S_0 and an upward-closed set U. Then, the following holds:

$$\mathsf{Reach}_{\mathcal{M}}(S_0) \subseteq dcone(\mathtt{CS}(\mathcal{M}, S_0)) \subseteq dcone(\mathtt{WCS}(\mathcal{M}, S_0, U))$$
$$\mathsf{Reach}_{\mathcal{M}}(S_0) \cap U \neq \emptyset \text{ if and only if } dcone(\mathtt{WCS}(\mathcal{M}, S_0, U)) \cap U \neq \emptyset$$

Proof. We prove $\mathsf{Reach}_{\mathcal{M}}(S_0) \subseteq dcone(\mathtt{CS}(\mathcal{M}, S_0)) \subseteq dcone(\mathtt{WCS}(\mathcal{M}, S_0, U))$ as follows. By definition, we have $\mathsf{Reach}_{\mathcal{M}}(S_0) \subseteq dcone(\mathtt{CS}(\mathcal{M}, S_0))$ (Definition 60(1)) and $\mathsf{Reach}_{\mathcal{M}}(S_0) \subseteq dcone(\mathtt{WCS}(\mathcal{M}, S_0, U))$ (Definition 65(1)). Moreover, a coverability set is a generator set of the smallest downward closed set containing $\mathsf{Reach}_{\mathcal{M}}(S_0)$ and $dcone(\mathtt{WCS}(\mathcal{M}, S_0, U))$ is a downward closed set containing $\mathsf{Reach}_{\mathcal{M}}(S_0)$. We conclude that $dcone(\mathtt{WCS}(\mathcal{M}, S_0, U))$ contains at least all the markings in $dcone(\mathtt{CS}(\mathcal{M}, S_0))$, i.e. $dcone(\mathtt{CS}(\mathcal{M}, S_0)) \subseteq dcone(\mathtt{WCS}(\mathcal{M}, S_0, U))$.

We prove that $\mathsf{Reach}_{\mathcal{M}}(S_0) \cap U \neq \emptyset$ if and only if $dcone(\mathtt{WCS}(\mathcal{M}, S_0, U)) \cap U \neq \emptyset$ as follows.

$\Rightarrow$ By definition (Definition 65(1)).

$\Leftarrow$ if $dcone(\mathtt{WCS}(\mathcal{M}, S_0, U)) \cap U \neq \emptyset$, there exists $\mathbf{m} \in \mathtt{WCS}(\mathcal{M}, S_0, U)$ such that $dcone(\mathbf{m}) \cap U \neq \emptyset$. We consider two cases. Either $\mathbf{m} \in \mathsf{Reach}_{\mathcal{M}}(S_0)$ and we have $\mathsf{Reach}_{\mathcal{M}}(S_0) \cap U \neq \emptyset$, or $\mathbf{m} \notin \mathsf{Reach}_{\mathcal{M}}(S_0)$. In the last case, there exists an infinite sequence $\mathcal{S} = \mathbf{m}_1\mathbf{m}_2\ldots$ with $\mathbf{m}_i \in \mathsf{Reach}_{\mathcal{M}}(S_0)$ and $\mathbf{m}_i \prec \mathbf{m}_{i+1}$ for all $i \geq 1$. Moreover, if $\mathbf{m}(p) \neq +\infty$, then $\mathbf{m}(p)$ is the biggest value of p in $\mathcal{S}$. Otherwise if $\mathbf{m}(p) = +\infty$, then there there exists $i \geq 1$ such that for all $i' \geq i$ we have $\mathbf{m}_{i'}(p) \geq \mathbf{m}_U(p)$. Hence, there exists $i \geq 1$ such that $\mathbf{m}_i(p) = \mathbf{m}(p)$ if $\mathbf{m}(p) \in \mathbb{N}$, and $\mathbf{m}_i(p) \geq \mathbf{m}_U(p)$ otherwise. Since $dcone(\mathbf{m}) \cap U \neq \emptyset$, we have $\mathbf{m}_U \preccurlyeq \mathbf{m}$. By construction, we have $\mathbf{m}_U \preccurlyeq \mathbf{m}_i$ and hence $\mathbf{m}_i \in U$. Since $\mathbf{m}_i \in \mathsf{Reach}_{\mathcal{M}}(S_0)$, we conclude that $\mathsf{Reach}_{\mathcal{M}}(S_0) \cap U \neq \emptyset$. □

However, it cannot be used to decide place boundedness.

Algorithm to Compute a WCS

In this section, we give an algorithm that decides the reachability problem of an upward closed set U for a MITN $\mathcal{M}$ and a set of initial marking S_0 by computing a weak coverability set of a MITN $\mathcal{M}$ for S_0 and U generated by the marking $\mathbf{m}_U$. That algorithm is Algorithm 21 and is presented as an extension of the enumerative algorithm 17. Notice that, to simplify the presentation, we did not make improvements similar to those of Algorithm 18. The algorithm is similar to Algorithm 17, except that it uses the acceleration function AccIsolated (line 4 of Algorithm 21) instead of the Karp and Miller acceleration function. Moreover, the arcs $\langle \mathbf{m}, \mathbf{m}' \rangle$ are labeled by the set of multi-transfers that have been used to construct $\mathbf{m}'$ from $\mathbf{m}$ (line 5 of Algorithm 21). Those sets are used when applying the acceleration function.

Algorithm 21: Extended_Forward_Algorithm

Input: $N = \langle \mathcal{P}, \mathcal{B} \rangle$: a MITN; S_0 : a set of initial markings; $\mathbf{m}_U$: the minimal marking of the upward closed set U (hence we consider that U is non empty)

Result: *boolean*

begin

  $F \leftarrow S_0$;

  $fixpoint \leftarrow S_0$;

  $T \leftarrow \emptyset$;

1 **while** $F \neq \emptyset$ **do**

    Choose some $\mathbf{m} \in F$;

    $F \leftarrow F \setminus \{\mathbf{m}\}$;

    foreach $M \in \mathcal{B}$ *such that* M *is firable from* $\mathbf{m}$ **do**

2   Let $\mathbf{n}$ be such that $\mathbf{m} \rightarrowtail^{M} \mathbf{n}$;

3   **if** $\nexists \mathbf{m}' \in fixpoint : \mathbf{n} \preccurlyeq \mathbf{m}'$ **then**

        $newlabel = \{M\}$;

        if $\exists \mathbf{m}' \in fixpoint$: $\mathsf{Path}(\mathbf{m}', \mathbf{m})$ *and* $\mathbf{m}' \prec \mathbf{n}$ **then**

          Let $\mathbf{m}'$ be such that $\mathsf{Path}(\mathbf{m}', \mathbf{m})$ and $\mathbf{m}' \prec \mathbf{n}$;

          Let $\langle \mathbf{m}', \mathbf{m}_1 \rangle \dots \langle \mathbf{m}_k, \mathbf{m} \rangle$ be the path from $\mathbf{m}'$ to $\mathbf{m}$;

          $newlabel = newlabel \cup \mathsf{Label}(\langle \mathbf{m}', \mathbf{m}_1 \rangle) \cup \dots \mathsf{Label}(\langle \mathbf{m}_k, \mathbf{m} \rangle)$;

4          $\mathbf{n} \leftarrow \mathsf{AccIsolated}(\mathbf{m}', \mathbf{n}, newlabel, \mathbf{m}_U)$;

5        $\mathsf{Label}(\langle \mathbf{m}, \mathbf{n} \rangle) = newlabel$;

        $T \leftarrow T \cup \{\langle \mathbf{m}, \mathbf{n} \rangle\}$;

        $fixpoint \leftarrow fixpoint \cup \{\mathbf{n}\}$;

6        $F \leftarrow F \cup \{\mathbf{n}\}$;

  if $\exists \mathbf{m} \in fixpoint : \mathbf{m}_U \preccurlyeq \mathbf{m}$ **then**

    return *true* ;

  else

    return *false* ;

end

More precisely, the acceleration function (called $\mathsf{AccIsolated}$) is defined as follows.

Definition 66 (The $\mathsf{AccIsolated}$ acceleration function) Let $\mathbf{m}$ and $\mathbf{m}'$ be two ω-markings such that $\mathbf{m} \prec \mathbf{m}'$, $\mathcal{T}$ be a set of multi-transfers and U be the upward closed set generated by the marking $\mathbf{m}_U$.

Then, the following holds.
$\mathsf{AccIsolated}(\mathbf{m}, \mathbf{m}', \mathcal{T}, U)$ returns the ω-marking $\mathbf{n}$ such that, for any place p, we have $\mathbf{n}(p) = +\infty$ **if** one of the following conditions holds.

(1) p is neither a source nor the target of a transfer in $\mathcal{T}$ and $\mathbf{m}(p) < \mathbf{m}'(p)$. More formally:

$$(\forall \langle P, p_t \rangle \in \mathsf{Transfer}(\mathcal{T}) : p \notin (\{p_t\} \cup P)) \text{ and } \mathbf{m}(p) < \mathbf{m}'(p)$$

(2) p is the target of a transfer in $\mathcal{T}$ and either $\mathbf{m}(p) < \mathbf{m}'(p)$ or there is a source p' for p of that transfer such that $\mathbf{m}(p') < \mathbf{m}'(p')$. More formally:

$$\exists \langle P, p_t \rangle \in \mathsf{Transfer}(\mathcal{T}) \text{ with } p = p_t \text{ and}$$

$$(\mathbf{m}(p) < \mathbf{m}'(p) \vee \exists p' \in P : \mathbf{m}(p') < \mathbf{m}'(p'))$$

(3) p is a source of a transfer in $\mathcal{T}$, $\mathbf{m}(p) < \mathbf{m}'(p)$, and furthermore $\mathbf{m}'(p) \geq \mathbf{m}_U(p)$. More formally:

$$\exists \langle P, p_t \rangle \in \mathsf{Transfer}(\mathcal{T}) \text{ with } p \in P \text{ and } \mathbf{m}(p) < \mathbf{m}'(p) \text{ and}$$

$$\mathbf{m}'(p) \geq \mathbf{m}_U(p)$$

otherwise, $\mathbf{n}(p) = \mathbf{m}'(p)$.

Notice that, since $\mathcal{T}$ will be a set of multi-transfers from a MITN, the cases (1)-(2)-(3) of Definition 66 are mutually exclusive. Remark also that, due to the case (3), $\mathsf{AccIsolated}$ will not always accelerate. Indeed, if all the places that increase between $\mathbf{m}$ and $\mathbf{m}'$ are source places of transfers in $\mathcal{T}$, but $\mathbf{m}'$ is lower than $\mathbf{m}_u$ for them, and their target places are already equal to $+\infty$, $\mathsf{AccIsolated}$ returns $\mathbf{m}'$. Moreover, it may happen that $\mathbf{m}'(p)$ is set to $+\infty$ but was already so.

Intuitively the acceleration function is justified as follows. Suppose that we have $\mathbf{m} \rightarrowtail^{\sigma} \mathbf{m}'$ and there exists a path in the tree from $\mathbf{m}$ to $\mathbf{m}'$ constructed by applying σ. Assume that the acceleration function is applied on $\mathbf{m}$ and $\mathbf{m}'$. So we have $\mathbf{m} \prec \mathbf{m}'$. The set of multi-transfers $\mathcal{T}$ given to $\mathsf{AccIsolated}$ is the set of multi-transfers used to construct $\mathbf{m}'$ from $\mathbf{m}$. In this case, $\mathcal{T}$ is the set of the multi-transfers that compose σ.

The only particularity of the $\mathsf{AccIsolated}$ function with respect to the Karp-Miller function is for places involved in a transfer. Let us take the case of a target place p. In the definition of the acceleration (Definition 66), we put $+\infty$ in two cases : (1) when p has strictly increased, or (2) one of its sources has strictly increased. Case (1) is classically justified by strict monotonicity of the model, i.e. repeating σ will add at least the same number of tokens in p each time (remember that p is not the source of any transfer). Case (2) is justified as follows: p is not the source of any transfer, so the sequence σ takes out of p a constant number of tokens each time it is fired. Furthermore, as a source of p strictly increased between $\mathbf{m}$ and $\mathbf{m}'$ and this place is not the source of a transfer with a target place different from p, when repeating σ, from $\mathbf{m}'$, we know that the target will increase strictly and then we simply apply the justification of case (1). For the sources, the justification is as follows. When a source p increases strictly between $\mathbf{m}$ and $\mathbf{m}'$, we know that it will not decrease when repeating σ. Actually, the number of tokens in each source places after firing σ is always the same, i.e. it is equal to the number of tokens added into those places after the last transfer of their tokens to their target place. So if its value is greater or equal to $\mathbf{m}_U(p)$, we know that it will stay so. By putting $+\infty$ in p, we do not take a too rough approximation for the following two reasons. First, putting $+\infty$ in p is safe w.r.t. the intersection with U, that is $\mathbf{n}(p) \geq \mathbf{m}_U(p)$ (where $\mathbf{n}$ is the result of $\mathsf{AccIsolated}$) if and only if all the markings $\mathbf{n}'$ constructed from $\mathbf{m}'$ by iterating σ are such that $\mathbf{n}' \geq \mathbf{m}_U(p)$. Second, it is easy to see that if we put $+\infty$ in a source p, then we also put a $+\infty$ in the target p_t of the transfers from p (which is unique following Definition 63), and this $+\infty$ will never leave the place p_t as we know that p_t is not the source of a transfer. As a consequence, the $+\infty$ in p is guaranteed not to "propagate" unsafely in the net.

Algorithm 21 manipulates weak limits of increasing sequences of reachable markings that are constructed by the acceleration function $\mathsf{AccIsolated}$. Those weak limits have a special form. More precisely, if $\mathbf{m}$ is a weak limit of the sequence $\mathcal{S}$ of markings, then $\mathbf{m}$ will have the following property. If $\mathcal{S}$ does not increase infinitely often for a place p, then $\mathbf{m}$ will have $+\infty$ in p only if p is a source place of a transfer and the sequence of reachable markings increases infinitely often for the target place of transfers from p (which is unique following the definition of MITN).

We call this type of weak limit *Isolated Weak Limit.*

Definition 67 (Isolated Weak Limit) Given a MITN $\mathcal{M} = \langle \mathcal{P}, \mathcal{B} \rangle$ and an upward closed set U of markings generated by the minimal marking $\mathbf{m}_U$, a marking $\mathbf{m}$ is an *isolated weak limit according to U (U-iwl)* of an infinite sequence $\mathcal{S} = \mathbf{m}_1\mathbf{m}_2 \ldots \mathbf{m}_n \ldots$ with $\mathbf{m}_i \prec \mathbf{m}_{i+1}$ for any $i \geq 1$ if $\mathbf{m}$ is an U-weak limit of $\mathcal{S}$ and moreover, if $\mathbf{l} = lub(\{\mathbf{m}_1, \mathbf{m}_2, \ldots\})$, for each place $p \in \mathcal{P}$ with $\mathbf{m}(p) = +\infty$ and $\mathbf{l}(p) \neq +\infty$, there is a multi-transfer $M \in \mathcal{B}$ with $\langle P, p_t \rangle \in \mathsf{Transfer}(M)$ such that $p \in P$ and $\mathbf{l}(p_t) = +\infty$.

In the following, given the MITN $\mathcal{M}$ with the initial markings S_0 and the upward closed set U generated by the minimal marking $\mathbf{m}_U$, we will say that $\mathbf{m}$ is an U-iwl if there exists a sequence $\mathcal{S} = \mathbf{m}_1\mathbf{m}_2 \ldots$ with $\mathbf{m}_i \in \mathsf{Reach}_{\mathcal{M}}(S_0)$ and $\mathbf{m}_i \prec \mathbf{m}_{i+1}$ for any $i \geq 1$ such that $\mathbf{m}$ is an U-iwl of $\mathcal{S}$.

Two important properties of Algorithm 21 need to be proved:

1. its termination;
2. its correctness, i.e. we need to prove that Algorithm 21 returns *true* if the given upward closed set U is reachable from S_0, *false* otherwise (because $fixpoint$ is a weak coverability set when the algorithm terminates (where the weak limits are iwl)).

Termination will be proved in Lemma 18.

To prove the correctness of the algorithm, we consider the pairs $(T_i, fixpoint_i)$ where T_i is the set of arcs of the tree contained into T after the ith iteration of the main loop (line 1), and $fixpoint_i$ is the set of ω-markings contained into $fixpoint$ after the ith iteration of the main loop (line 1). By convention, T_0 and $fixpoint_0$ are the initial value of T and $fixpoint$, i.e. their contents before the first iteration (hence after a fictitious $0th$ iteration). Notice that the following lemma holds.

Lemma 17 *$\forall i \geq 0$, we have $\mathbf{n} \in fixpoint_i$ if and only if $\mathbf{n} \in S_0$ or there exists $\langle \mathbf{m}, \mathbf{n} \rangle \in T_i$ with $\mathbf{m} \in fixpoint_{i-1}$.*

Proof. This immediately results from the initialisations in Algorithm 21 (before line 1) and from the updates (before line 6), noticing that $F_i \subseteq fixpoint_i$. □

The proof of the correctness of Algorithm 21 will be cut into two parts (assuming that it terminates):

(i) First, we prove that Algorithm 21 returns *true* if the upward closed set U (described by the minimal marking $\mathbf{m}_U$) is reachable from S_0 (Lemma 19);

(ii) second, we prove that Algorithm 21 returns *false* if the upward closed set U (described by the marking point $\mathbf{m}_U$) is not reachable from S_0 (Lemma 23).

Point (i) consists in proving that if Algorithm 21 terminates after k iterations of the main loop, then we have $\mathbf{m} \in dcones(fixpoint_k)$ for all the reachable markings $\mathbf{m}$ (see Lemma 19). Indeed, assuming this is true, if U is reachable from S_0, then there is $\mathbf{m}' \in fixpoint_k$ with $\mathbf{m}_U \preccurlyeq \mathbf{m}'$ and the algorithm returns *true*. Point (ii) needs intermediate results on the pairs $(T_i, fixpoint_i)$ to be proved. More precisely, we shall prove (see Lemma 22) that, after k iterations of the main loop in Algorithm 21 we can associate to each marking $\mathbf{m} \in fixoint_k$ an infinite sequence $\mathcal{S}^{\mathbf{m}} = \mathbf{m}_1\mathbf{m}_2\ldots$ of reachable markings from S_0 such that $\mathbf{m}_j \preceq \mathbf{m}_{j+1}$ for all $j \geq 1$ and

C_1 either $\mathbf{m} = \mathbf{m}_j$ for all $j \geq 1$ (if $\mathbf{m}$ is a reachable marking) or $\mathcal{S}^{\mathbf{m}}$ is a strictly increasing sequence and $\mathbf{m}$ is an U-iwl of $\mathcal{S}^{\mathbf{m}}$;

C_2 if we have $\langle \mathbf{m}, \mathbf{m}' \rangle \in T_i$, there exists an infinite sequence $\sigma_1\sigma_2\ldots$ where σ_i ($i \geq 1$) is a sequence of multi-transfers in $\mathsf{Label}(\langle \mathbf{m}, \mathbf{m}' \rangle)$ (and containing at least one occurrence of each multi-transfer in $\mathsf{Label}(\langle \mathbf{m}, \mathbf{m}' \rangle)$) that labels the arc $\langle \mathbf{m}, \mathbf{m}' \rangle$ (see Algorithm 21) such that $\mathbf{m}_i \rightarrowtail^{\sigma_i} \mathbf{m}'_i$ where $\mathcal{S}^{\mathbf{m}'} = \mathbf{m}'_1\mathbf{m}'_2\ldots$ is an infinite sequence associated to $\mathbf{m}'$ with the same properties (recursively) and so can be constructed from $\mathcal{S}^{\mathbf{m}}$ in a systematical way.

The condition C_1 means that each ω-marking into *fixpoint* either is a reachable marking or weakly covers an infinite strictly increasing sequence of reachable markings. The condition C_2 means that the relation defined by the tree T between the ω-markings in *fixpoint* covers (a part of) the accessibility relation on the reachable markings covered by those ω-markings.

In the following, we can always suppose that for each place p we have $\mathbf{m}_j(p) = \mathbf{m}_{j+1}(p)$ $\forall j \geq 1$ or $\mathbf{m}_j(p) < \mathbf{m}_{j+1}(p) \forall j \geq 1$. Indeed, if $I \subseteq \mathbb{N}$ such that $\mathbb{N} \setminus I$ is infinite and we remove in all the sequences $\mathcal{S}^{\mathbf{m}}$ the markings $\mathbf{m}_i$ where $i \in I$, we preserve C_1 and C_2. Moreover, we can choose I such that the new sequences constructed in the proofs satisfy the property.

We shall proceed by induction on the number of iterations of the main loop (line 1 of Algorithm 21). In the base case, the two points are trivially verified, as $fixpoint_0 = S_0$ and $T_0 = \emptyset$, with $\mathcal{S}^{\mathbf{m}} = \mathbf{m}\mathbf{m}\ldots$ for each $\mathbf{m} \in S_0$. If we make the hypothesis that the two properties hold for $(T_i, fixpoint_i)$ with i such that $1 \leq i \leq n < k$, it remains us to prove that the properties holds for $(T_{n+1}, fixpoint_{n+1})$. At the $(n+1)th$ iteration, each new ω-marking computed at line 2 is either forgotten (line 3), or directly added into the tree or produced by the acceleration function. The first case does not modify T and *fixpoint* and lemma 20 states that the second case ensures that C_1 and C_2 hold when adding ω-markings (the case where a reachable marking is added is trivial). Lemma 22 states that all the cases, and hence the third case, ensure that C_1 and C_2 hold.

This allows us to conclude that the two properties are verified by $(T_{n+1}, fixpoint_{n+1})$ and hence is verified by any $(T_i, fixpoint_i)$ with $1 \leq i \leq k$. So, at the end of the algorithm, all the ω-markings are either $U - iwls$ (hence U-weak limits) or reachable markings. This will allow us to conclude from Proposition 20 that if $\mathsf{Reach}_{\mathcal{M}}(S_0) \cap U = \emptyset$, then Algorithm 21 returns $false$. We now go into the details of the proof for each point.

Termination. To prove termination, we first need the following property of $\preccurlyeq$ on $\mathbb{N} \cup \{+\infty\}$.

Proposition 21 Let $\mathbf{m}_1\mathbf{m}_2\ldots$ be an infinite sequence of ω-markings of dimension k. Then, there exists an infinite sub-sequence $\mathbf{m}_{i_1}\mathbf{m}_{i_2}\ldots$ such that either $\mathbf{m}_{i_j} = \mathbf{m}_{i_{j+1}}$ for any $j \geq 1$ or $\mathbf{m}_{i_j} \prec \mathbf{m}_{i_{j+1}}$ for any $j \geq 1$ (i.e. either constant or strictly increasing).

Proof. We exhibit an inductive reasoning on the dimension of the markings.

Base case: Let $k = 1$. Then $\mathbf{m}_1\mathbf{m}_2\ldots$ is a sequence of elements in $\mathbb{N} \cup \{\omega\}$. If one of its components occurs infinitely often, the property is trivially true. Otherwise, let min be the minimal value in this sequence and i_1 be its last index, then we take $\mathbf{m}_{i_1}$ as the first element and we iterate the construction on $\mathbf{m}_{i_1+1}\mathbf{m}_{i_1+2}\ldots$ We then obtain the desired sequence $\mathbf{m}_{i_1}\mathbf{m}_{i_2}\ldots$

Induction step: By induction hypothesis, we know that for any sequence of markings in $(\mathbb{N} \cup \{+\infty\})^{k-1}$, we can extract an infinite constant or strictly increasing sub-sequence of ω-markings. Let $\mathbf{m}_1\mathbf{m}_2\ldots$ be a sequence of ω-markings in $(\mathbb{N} \cup \{+\infty\})^k$ and $\mathbf{m}_{i_1}\mathbf{m}_{i_2}\ldots$ be an infinite sub-sequence such that it is either constant or strictly increasing for the $k-1$ first dimensions. Such a sub-sequence exists by induction hypothesis. By applying the procedure of the base case, we can extract a sub-sequence of it which is either constant or strictly increasing for the k dimensions. □

Termination of Algorithm 21 is stated by the following lemma.

Lemma 18 Algorithm 21 terminates.

Proof. Suppose that Algorithm 21 does not terminate for some input parameters. In this case, Algorithm 21 adds infinitely often new ω-markings in the tree, since otherwise F decreases to the empty set. We consider two cases.

- The algorithm applies the acceleration function a finite number of times. Let us consider the infinite sequence $T_1, T_2, \ldots$ of the trees obtained at the end of the successive iterations of the main loop (line 1). As each tree T_i contains T_{i-1} and is finitely branching, there is a branch that increases infinitely often in this sequence of trees. Let $\mathbf{m}_1, \mathbf{m}_2, \ldots$ be the sequence of ω-markings corresponding to this path. Let l be the iteration of the main loop from which Algorithm 21 does not apply $\mathsf{AccIsolated}$ anymore. As $\preccurlyeq$ is a well-quasi ordering (see Lemma 9), there exists $i, j \in \mathbb{N}$ with $l < i < j$ such that $\mathbf{m}_i \preccurlyeq \mathbf{m}_j$. Moreover, $\mathbf{m}_j$ is constructed after the lth iteration of the main loop of line 1. As the algorithm never adds ω-markings that are less or equal to ω-markings in $fixpoint$, as $\mathbf{m}_i \in fixpoint_{l'}$ where $l' \geq l$ and T_l is the first tree containing $\mathbf{m}_i$ (see Lemma 17), and since $\mathbf{m}_j$ is added into the tree after $\mathbf{m}_i$, we have that $\mathbf{m}_i \prec \mathbf{m}_j$. We conclude that

AccIsolated is applied during the construction of $\mathbf{m}_j$. That makes a contradiction since $\mathbf{m}_j$ is constructed after the lth iteration of the main loop of Algorithm 21;

- The algorithm applies the acceleration function an infinite number of times. Let us consider the infinite sequence $T_1, T_2, \ldots$ of the trees obtained at the end of the successive iterations of the main loop (line 1). As observed above, there is a branch that increases infinitely often in the sequence $T_1, T_2, \ldots$ Let $\mathcal{S} = \mathbf{m}_1, \mathbf{m}_2, \ldots$ be the sequence of ω-markings corresponding to this branch. As $\preccurlyeq$ is a well-quasi ordering (see Lemma 9), from Proposition 21 we have that there exists an infinite sub-sequence $\mathcal{S}' = \mathbf{m}_{i_1}\mathbf{m}_{i_2}\ldots$ of $\mathcal{S}$ such that $\mathbf{m}_{i_j} \preccurlyeq \mathbf{m}_{i_{j+1}}$ for all $j \geq 1$. Moreover, as the algorithm never adds ω-markings that are less or equal to ω-markings in the tree and since $\mathbf{m}_{i_{j+1}}$ is added into the tree after $\mathbf{m}_{i_j}$, we have that $\mathbf{m}_{i_j} \prec \mathbf{m}_{i_{j+1}}$ for all $j \geq 1$. So, AccIsolated is applied each time that $\mathbf{m}_{i_j}$ $(j > 1)$ is added.

  Let $Inf(\mathbf{m})$ be the number of dimensions equals to $+\infty$ in $\mathbf{m}$. As Inf is bounded by $\mid \mathcal{P} \mid$, then there exists $j \geq 1$ and $0 \leq k \leq \mid \mathcal{P} \mid$ such that for all $j' \geq j$ we have $Inf(\mathbf{m}_{i_{j'}}) = k$. This means that AccIsolated does not put new $+\infty$ when it constructs the ω-markings $\mathbf{m}_{i_j}, \mathbf{m}_{i_{j+1}}, \ldots$ As $\mathcal{S}'$ is a strictly increasing sequence of ω-markings, there is $p \in \mathcal{P}$ such that the number of tokens in p increases infinitely often in $\mathcal{S}'$. Hence, there is $l' \geq l$ such that $\mathbf{m}_{i_{l'}}(p) < \mathbf{m}_{i_{l'+1}}(p)$ with $\mathbf{m}_{i_{l'+1}}(p) \geq \mathbf{m}_U(p)$. By definition, AccIsolated always puts a new $+\infty$ in this case into p during the construction of $\mathbf{m}_{i_{l'+1}}$. That makes a contradiction with the fact that AccIsolated does not.

In both cases, we obtain a contradiction and we conclude that Algorithm 21 always terminates. □

Point $2-(i)$ Before proving Point $2-(i)$, we extend Proposition 1 to the case of ω-markings.

Proposition 22 When the markings are extended to ω-markings, all the transition relations defined by a MTN $\mathcal{M} = \langle \mathcal{P}, \mathcal{B} \rangle$ are monotonic with respect to the component-wise ordering over tuples in $(\mathbb{N} \cup \{+\infty\})^{|\mathcal{P}|}$.

Proof. The proof is similar to the proof of Proposition 1. □

Notice that in the case of ω-markings, MTNs are not strictly monotonic. Indeed, consider the ω-markings $\langle +\infty, 1\rangle$ and $\langle +\infty, 2\rangle$, and the transfer of tokens from the first place to the second one. by applying the transfer on $\langle +\infty, 1\rangle$, we obtain $\langle 0, +\infty\rangle$ and by applying the transfer from $\langle +\infty, 2\rangle$, we also obtain $\langle 0, +\infty\rangle$ but $\langle +\infty, 1\rangle \prec \langle +\infty, 2\rangle$.

The following lemma states that Algorithm 21 returns *true* when the upward closed set U (described by the minimal marking $\mathbf{m}_U$) is reachable from S_0.

Lemma 19 Given a MITN $\mathcal{M}$ with the set of initial markings S_0 and the upward-closed set U described by the minimal marking $\mathbf{m}_U$, Algorithm 21 returns *true* if U is reachable from S_0 in $\mathcal{M}$.

Proof. We prove that if the algorithm terminates after the kth iteration of the main loop, then for all reachable marking $\mathbf{m}$ from S_0, there exists $\mathbf{m}' \in fixpoint_k$ with $\mathbf{m} \preccurlyeq \mathbf{m}'$. We exhibit an inductive proof on the size of the smallest path from S_0 to $\mathbf{m}$.

Base case: Suppose that $\mathbf{m} \in S_0$. As $fixpoint_i \subseteq fixpoint_{i+1}$ for all i such that $0 \leq i \leq k$, we have that $\mathbf{m} \in fixpoint_0 \subseteq fixpoint_k$.

Induction step: Let us make the hypothesis that $\mathbf{m}$ is reachable in n steps and the last step is $\mathbf{m}' \rightarrowtail^M \mathbf{m}$. By induction hypothesis, $\mathbf{m}'$ is such that there exists $\mathbf{m}'' \succcurlyeq \mathbf{m}'$ and $\mathbf{m}'' \in fixpoint_k$. By monotonicity (see Proposition 22), $\mathbf{m}'' \rightarrowtail \mathbf{m}'''$ with $\mathbf{m}''' \succcurlyeq \mathbf{m}$. Notice that when $\mathbf{m}''$ is put into $fixpoint$, it is also put into F and hence Algorithm 21 will compute its successors and try to put them into $fixpoint$. If $\mathbf{m}''' \in fixpoint_k$, we are done; if $\mathbf{m}''' \notin fixpoint_k$, this is only possible if, when computing $\mathbf{m}'''$, there is already a ω-marking $\mathbf{m}''''$ in $fixpoint$ such that $\mathbf{m}'''' \succcurlyeq \mathbf{m}'''$ or the acceleration function is applied. In this last case, a ω-marking $\mathbf{m}'''' \succcurlyeq \mathbf{m}'''$ is added into $fixpoint$. In both cases, since $fixpoint_i \subseteq fixpoint_{i+1}$ for all i such that $0 \leq i \leq k$, we have $\mathbf{m}'''' \in fixpoint_k$.

Hence, if U is reachable from S_0, there exist $\mathbf{m}' \in \mathsf{Reach}_{\mathcal{M}}(S_0)$ such that $\mathbf{m}_U \preccurlyeq \mathbf{m}'$ and $\mathbf{m} \in fixpoint_k$ with $\mathbf{m}_U \preccurlyeq \mathbf{m}' \preccurlyeq \mathbf{m}$, and Algorithm 21 returns true. □

Notice that we did not need here to use the fact that $\mathcal{M}$ is isolated, hence property remains true for any MTN.

Point $2-(ii)$ We prove that if Algorithm 21 terminates after k iterations of the main loop, we can associate to each marking $\mathbf{m} \in fixoint_k$ an infinite sequence $\mathcal{S}^{\mathbf{m}} = \mathbf{m}_1\mathbf{m}_2\ldots$ of markings reachable from S_0 with $\mathbf{m}_j \preceq \mathbf{m}_{j+1}$ for all $j \geq 1$, such that

C_1 either $\mathbf{m} = \mathbf{m}_j$ for all $j \geq 1$ (i.e. $\mathbf{m}$ is a reachable marking) or $\mathcal{S}^{\mathbf{m}}$ is a strictly increasing sequence and $\mathbf{m}$ is an U-iwl of $\mathcal{S}^{\mathbf{m}}$;

C_2 if we have $\langle \mathbf{m}, \mathbf{m}' \rangle \in T_i$, there exists an infinite sequence $\sigma_1\sigma_2\ldots$ where σ_i ($i \geq 1$) is a sequence of multi-transfers in $\mathsf{Label}(\langle \mathbf{m}, \mathbf{m}' \rangle)$ (and containing at least one occurrence of each multi-transfer in $\mathsf{Label}(\langle \mathbf{m}, \mathbf{m}' \rangle)$) that labels the arc $\langle \mathbf{m}, \mathbf{m}' \rangle$ (see Algorithm 21) such that $\mathbf{m}_i \rightarrowtail^{\sigma_i} \mathbf{m}'_i$ and $\mathcal{S}^{\mathbf{m}'} = \mathbf{m}'_1\mathbf{m}'_2\ldots$ is the infinite sequence associated to $\mathbf{m}'$ with the requested properties and so can be constructed from $\mathcal{S}^{\mathbf{m}}$ in a systematical way.

As indicated above, we have to establish the following lemma.

Lemma 20 Let $\mathcal{M} = \langle \mathcal{P}, \mathcal{B} \rangle$ be a MITN with the set of initial markings S_0, and let U be an upward closed set of markings generated by $\mathbf{m}_U$. Let $\mathbf{m}$ be an U-iwl of the sequence of markings $\mathbf{m}_1\mathbf{m}_2\ldots$ with $\mathbf{m}_i \prec \mathbf{m}_{i+1}$ and $\mathbf{m}_i \in \mathsf{Reach}_{\mathcal{M}}(S_0)$ for all $i \geq 1$. Then, for any $M \in \mathcal{B}$, if we have $\mathbf{m} \rightarrowtail^M \mathbf{n}$, then $\mathbf{n}$ is an U-iwl of a strictly increasing infinite sequence of markings $\mathbf{n}_1\mathbf{n}_2\ldots$ such that there is $j \geq 1$ with $\mathbf{m}_{j+i} \rightarrowtail^M \mathbf{n}_i$ for any $i \geq 1$.

Proof. Let $M = \langle\langle \mathcal{I}, \mathcal{O} \rangle, B \rangle$. We assume that $M_{\mathcal{I}} = \langle\langle \mathcal{I}, \mathcal{O}_0 \rangle, \emptyset \rangle$, $M_B = \langle\langle \mathcal{I}_0, \mathcal{O}_0 \rangle, B \rangle$ and $M_{\mathcal{O}} = \langle\langle \mathcal{I}_0, \mathcal{O} \rangle, \emptyset \rangle$. Remember that $\mathcal{I}_0$ and $\mathcal{O}_0$ are such that $\mathcal{I}_0(p) = 0$ and $\mathcal{O}_0(p) = 0$ for all the places p. From Definition 11, we have

$$\mathbf{m} \rightarrowtail^M \mathbf{n} \text{ iff } \exists \mathbf{m}^{\mathcal{I}}, \mathbf{m}^B : \mathbf{m} \rightarrowtail^{M_{\mathcal{I}}} \mathbf{m}^{\mathcal{I}} \rightarrowtail^{M_B} \mathbf{m}^B \rightarrowtail^{M_{\mathcal{O}}} \mathbf{n}$$

First, let $\mathbf{l} = lub(\{\mathbf{m}_1, \mathbf{m}_2, \ldots\})$. By definition of an isolated weak limit and condition (iii) of Definition 63, for each place p such that $\mathbf{m}(p) = +\infty$ and $\mathbf{l}(p) \neq +\infty$ then p is the source of a transfer (but not necessarily the source of a transfer of M) and so $\mathcal{I}(p) = 0$; otherwise $\mathbf{m}(p) = \mathbf{l}(p)$. It follows that $M^{\mathcal{I}}$ is firable from $\mathbf{l}$, hence there exists $i \geq 1$ such that for all

$j \geq i$ we have that $M^{\mathcal{I}}$ is firable from $\mathbf{m}_j$ and we can construct a sequence $\mathbf{m}_1^{\mathcal{I}}\mathbf{m}_2^{\mathcal{I}}\ldots$ as follows:

$$\begin{array}{ccccc} \mathbf{m}_i & \prec & \mathbf{m}_{i+1} & \prec & \ldots \\ \downarrow_{M_{\mathcal{I}}} & & \downarrow_{M_{\mathcal{I}}} & & \ldots \\ \mathbf{m}_1^{\mathcal{I}} & \prec & \mathbf{m}_2^{\mathcal{I}} & \prec & \ldots \end{array}$$

Let $\mathbf{l}^{\mathcal{I}} = lub(\{\mathbf{m}_1^{\mathcal{I}}, \mathbf{m}_2^{\mathcal{I}}, \ldots\})$. As the effect of a Petri net transition is constant, we have

$$\forall p, \forall j \geq 1 : \mathbf{m}_{i+j-1}(p) < \mathbf{m}_{i+j}(p) \text{ if and only if } \mathbf{m}_j^{\mathcal{I}}(p) < \mathbf{m}_{j+1}^{\mathcal{I}}(p)$$

so that $\mathbf{l}(p) = +\infty$ if and only if $\mathbf{l}^{\mathcal{I}}(p) = +\infty$. For such places p, we also have by construction $\mathbf{m}(p) = \mathbf{m}^{\mathcal{I}}(p) = +\infty$. In any case, for any place p, we have $\mathbf{m}_j^{\mathcal{I}}(p) = \mathbf{m}_{j+i-1}(p) - \mathcal{I}(p) \forall j \geq 1$ so that $\mathbf{l}^{\mathcal{I}}(p) = \mathbf{l}(p) - \mathcal{I}(p)$, and $\mathbf{m}^{\mathcal{I}}(p) = \mathbf{m}(p) - \mathcal{I}(p)$. Hence, $\mathbf{m}^{\mathcal{I}}$ is an U-iwl of the infinite increasing sequence $\mathbf{m}_1^{\mathcal{I}}\mathbf{m}_2^{\mathcal{I}}\ldots$ because for all the places p where $\mathbf{m}^{\mathcal{I}}(p) \neq +\infty$ we have that $+\infty \neq \mathbf{m}(p) = \mathbf{l}(p)$ and $\mathbf{m}^{\mathcal{I}}(p) = \mathbf{l}^{\mathcal{I}}(p)$, and for all the places p such that $\mathbf{m}^{\mathcal{I}}(p) = +\infty$ we have that $\mathbf{l}^{\mathcal{I}}(p) \geq \mathbf{m}_U(p)$ since either $\mathbf{l}^{\mathcal{I}}(p) = +\infty$ or $\mathbf{l}^{\mathcal{I}}(p) \neq +\infty \neq \mathbf{l}(p)$, $\mathbf{m}(p) = +\infty$ and $\mathcal{I}(p) = 0$ so that $\mathbf{l}^{\mathcal{I}}(p) = \mathbf{l}(p) \geq \mathbf{m}_U(p)$. Moreover, if $\mathbf{m}^{\mathcal{I}}(p) = +\infty$ but $\mathbf{l}^{\mathcal{I}}(p) \neq +\infty$, then p is a source place of a transfer and its target place p', by definition, is such that $\mathbf{l}(p') = +\infty = \mathbf{l}^{\mathcal{I}}(p)$.

Second, let us define the infinite sequence of markings $\mathbf{m}_1^B\mathbf{m}_2^B\ldots$ as follows.

$$\begin{array}{ccccc} \mathbf{m}_1^{\mathcal{I}} & \prec & \mathbf{m}_2^{\mathcal{I}} & \prec & \ldots \\ \downarrow_{M_B} & & \downarrow_{M_B} & & \ldots \\ \mathbf{m}_1^B & \prec & \mathbf{m}_2^B & \prec & \ldots \end{array}$$

Let $\mathbf{l}^B = lub(\{\mathbf{m}_1^B, \mathbf{m}_2^B, \ldots\})$. For any place p that is neither a source nor a target of a transfer in M_B, we have $\mathbf{m}_j^B(p) = \mathbf{m}_j^{\mathcal{I}}(p) \forall j \geq 1$, $\mathbf{l}^{\mathcal{I}}(p) = \mathbf{l}^B(p)$ and $\mathbf{m}^{\mathcal{I}}(p) = \mathbf{m}^B(p)$. Moreover, for any $\langle P, p\rangle \in \mathsf{Transfer}(M_B)$, we have by construction that $\mathbf{m}_j^B(P) = 0 \forall j \geq 1$, $\mathbf{l}^B(P) = 0$ and $\mathbf{m}^B(P) = 0$, as source places of transfers cannot be target places of transfers in M_B and so after the transfer those places are empty. This is a direct implication of the conditions 1. and 2. of Definition 9.

For each target place p of a transfer in M, i.e. for each place p such that there exists $\langle P, p_t\rangle \in \mathsf{Transfer}(M)$ with $p_t = p$, we have that p cannot be a source of a multi-transfer (property (ii) of the Definition 63), $\mathbf{m}^B(p) = \mathbf{m}^{\mathcal{I}}(p) + \sum_{p' \in \mathsf{Sources}(M,p)} \mathbf{m}^{\mathcal{I}}(p')$ and $\mathbf{m}_i^B(p) = \mathbf{m}_i^{\mathcal{I}}(p) + \sum_{p' \in \mathsf{Sources}(M,p)} \mathbf{m}_i^{\mathcal{I}}(p') \forall i \geq 1$. Hence, the following holds.

- If $\mathbf{m}_i^{\mathcal{I}}(p) < \mathbf{m}_{i+1}^{\mathcal{I}}(p)$ for some $i \geq 1$, $\mathbf{m}_i^{\mathcal{I}}(p') \leq \mathbf{m}_{i+1}^{\mathcal{I}}(p') \forall i \geq 1, \forall p' \in \mathsf{Sources}(M,p)$. It follows that $\mathbf{m}_i^B(p) < \mathbf{m}_{i+1}^B(p)$. As a consequence, if $\mathbf{m}_i^{\mathcal{I}}(p) < \mathbf{m}_{i+1}^{\mathcal{I}}(p)$ for infinitely many $i's$, we have that $\mathbf{m}^{\mathcal{I}}(p) = \mathbf{l}^{\mathcal{I}}(p) = +\infty = \mathbf{m}^B(p) = \mathbf{l}^B(p)$;

- If, for infinitely many $i's$, $\mathbf{m}_i^{\mathcal{I}}(p) = \mathbf{m}_{i+1}^{\mathcal{I}}(p)$ but there exists $p' \in \mathsf{Sources}(M,p)$ such that $\mathbf{m}_i^{\mathcal{I}}(p') < \mathbf{m}_{i+1}^{\mathcal{I}}(p')$, we have that $\mathbf{m}_i^B(p) < \mathbf{m}_{i+1}^B(p)$ for all those $i's$ and $\mathbf{l}^B(p) = +\infty$. Moreover, for some $p' \in \mathsf{Sources}(M,p)$ we have $\mathbf{m}_i^{\mathcal{I}}(p') < \mathbf{m}_{i+1}^{\mathcal{I}}(p')$ for infinitely many $i's$ so that $\mathbf{m}^{\mathcal{I}}(p') = \mathbf{l}^{\mathcal{I}}(p') = +\infty$ and $\mathbf{m}^B(p) = +\infty$;

- If for any $i \geq j$, $\mathbf{m}_i^{\mathcal{I}}(p) = \mathbf{m}_{i+1}^{\mathcal{I}}(p) = \mathbf{l}^B(p) < +\infty$ and for all $p' \in \mathsf{Sources}(M,p)$ we have $\mathbf{m}_i^{\mathcal{I}}(p') = \mathbf{m}_{i+1}^{\mathcal{I}}(p') = \mathbf{l}^{\mathcal{I}}(p') < +\infty$, it follows that $\mathbf{m}_i^B(p) = \mathbf{m}_{i+1}^B(p)$ for any $i \geq j$, $\mathbf{l}^{\mathcal{I}}(p) = \mathbf{m}^{\mathcal{I}}(p)$ (since p is a target, hence not a source ((ii) of Definition 63), and $\mathbf{m}^{\mathcal{I}}$ is an $U - iwl$), $\mathbf{l}^{\mathcal{I}}(p') = \mathbf{m}^{\mathcal{I}}(p')$ for all $p' \in \mathsf{Sources}(M,p)$ (otherwise, since $\mathbf{m}^{\mathcal{I}}$ is an $U - iwl$ and p is the only target from p' in a MITN, we would have $\mathbf{l}^{\mathcal{I}}(p) = +\infty$) and

$\mathbf{m}^B(p) = \mathbf{m}^{\mathcal{I}}(p) + \sum_{p' \in \mathsf{Sources}(M,p)} \mathbf{m}^{\mathcal{I}}(p') = \mathbf{l}^{\mathcal{I}}(p) + \sum_{p' \in \mathsf{Sources}(M,p)} \mathbf{l}^{\mathcal{I}}(p') = \mathbf{m}_j^{\mathcal{I}}(p) + \sum_{p' \in \mathsf{Sources}(M,p)} \mathbf{m}_j^{\mathcal{I}}(p') = \mathbf{m}_j^B = \mathbf{l}^B(p)$.

We conclude that $\mathbf{m}^B$ is an U-iwl of the infinite sequence $\mathbf{m}_1^B\mathbf{m}_2^B\ldots$ with $\mathbf{m}_i^B \prec \mathbf{m}_{i+1}^B$ for all $i \geq 1$ because for each place p we have $\mathbf{m}^B(p) = \mathbf{l}^B(p)$ unless p is in the first case above (p is neither a source nor a target of transfer in M_B) where $\mathbf{l}^{\mathcal{I}}(p) = \mathbf{l}^B(p)$ and $\mathbf{m}^{\mathcal{I}}(p) = \mathbf{m}^B(p)$ so that the condition on $\mathbf{l}^B$ and $\mathbf{m}^B$ is satisfied since it is satisfied on $\mathbf{l}^{\mathcal{I}}$ and $\mathbf{m}^{\mathcal{I}}$.

Finally, we construct $\mathbf{n}_1\mathbf{n}_2\ldots$ as follows.

$$\begin{array}{ccccc} \mathbf{m}_1^B & \prec & \mathbf{m}_2^B & \prec & \ldots \\ \downarrow_{M_{\mathcal{O}}} & & \downarrow_{M_{\mathcal{O}}} & & \ldots \\ \mathbf{n}_1 & \prec & \mathbf{n}_2 & \prec & \ldots \end{array}$$

As $M^{\mathcal{O}}$ has constant effect, we can have a similar reasoning than in the case of the construction of $\mathbf{m}^{\mathcal{I}}$ from $\mathbf{m}$. We conclude that $\mathbf{n}$ is an U-iwl of the sequence $\mathbf{n}_1\mathbf{n}_2\ldots$ with $\mathbf{n}_i \prec \mathbf{n}_{i+1}$ for all $i \geq 1$. Moreover, we have that $\mathbf{n}_i \in \mathsf{Reach}_{\mathcal{M}}(S_0)$ for all $i \geq 1$ since we have $\mathbf{m}_i \in \mathsf{Reach}_{\mathcal{M}}(S_0)$. □

We extend Lemma 20 to sequences of multi-transfers.

Corollary 14 Let $\mathcal{M} = \langle \mathcal{P}, \mathcal{B} \rangle$ be a MITN with the set of initial markings S_0, and let U be an upward closed set of markings generated by $\mathbf{m}_U$. Let $\mathbf{m}$ be an U-iwl of the sequence of markings $\mathbf{m}_1\mathbf{m}_2\ldots$ with $\mathbf{m}_i \prec \mathbf{m}_{i+1}$ and $\mathbf{m}_i \in \mathsf{Reach}_{\mathcal{M}}(S_0)$ for all $i \geq 1$. Then, for any sequence σ of multi-transfers of $\mathcal{M}$, if we have $\mathbf{m} \rightarrowtail^{\sigma} \mathbf{n}$, then $\mathbf{n}$ is an U-iwl of a strictly increasing infinite sequence of markings $\mathbf{n}_1\mathbf{n}_2\ldots$ such that there is $j \geq 1$ with $\mathbf{m}_{j+i} \rightarrowtail^{\sigma} \mathbf{n}_i$ for any $i \geq 1$.

To prove that the acceleration function does not violate the two conditions C_1 and C_2, we need the following lemma.

Lemma 21 Let $\mathbf{m}_1$ and $\mathbf{m}_2$ be two markings, σ be a sequence of multi-transfers of a MITN and p a place such that there exists $\langle P, p_t \rangle \in \mathsf{Transfer}(\sigma)$ with $p_t = p$ (p is a target place of a transfer in σ). Then, the following holds.
If σ is firable from $\mathbf{m}_1$ and $\mathbf{m}_2$, and $\mathbf{m}_1(\mathsf{Source}(\sigma,p)) \leq \mathbf{m}_2(\mathsf{Source}(\sigma,p))$, then $\mathsf{Effect}(\sigma,\mathbf{m}_2,p) = \mathsf{Effect}(\sigma,\mathbf{m}_1,p) + \mathbf{m}_2(\mathsf{Sources}(\sigma,p)) - \mathbf{m}_1(\mathsf{Sources}(\sigma,p))$.

Proof. We exhibit an inductive reasoning on the number of transfers $\langle P, p_t \rangle \in \mathsf{Transfer}(\sigma)$ such that $p_t = p$.

Base case: There is one transfer $\langle P, p_t \rangle \in \mathsf{Transfer}(\sigma)$ such that $p_t = p$, so that $\mathsf{Sources}(\sigma,p) = P$.

For any place p', let $\mathcal{I}_\sigma(p')$ be the number of tokens removed from the place p' by Petri nets arcs when firing the sequence of multi-transfer σ and $\mathcal{O}_\sigma(p')$ the number of tokens added to the place p' by Petri nets arcs when firing σ. We decompose σ into the sequence $\sigma_1 \cdot M \cdot \sigma_2$ such that M is the multi-transfer of σ containing one transfer $\langle P, p_t \rangle$ with $p_t = p$. Since p is a target in a MITN, it is not a source ((ii) of Definition 63), and the places in P are not target nor sources of Petri net arcs ((iii) of Definition 63).

Suppose that we have $\mathbf{m}_1 \rightarrowtail^{\sigma} \mathbf{m}'_1$ and $\mathbf{m}_2 \rightarrowtail^{\sigma} \mathbf{m}'_2$. Following the definition of MITN, we have

$$\mathbf{m}'_1(p) = \mathbf{m}_1(p) + \mathcal{O}_\sigma(p) - \mathcal{I}_\sigma(p) + \mathbf{m}_1(\mathsf{Sources}(\sigma, p)) + \sum_{p' \in \mathsf{Sources}(\sigma_1, p)} \mathcal{O}_\sigma(p')$$

and

$$\mathbf{m}'_2(p) = \mathbf{m}_2(p) + \mathcal{O}_\sigma(p) - \mathcal{I}_\sigma(p) + \mathbf{m}_2(\mathsf{Sources}(\sigma, p)) + \sum_{p' \in \mathsf{Sources}(\sigma_1, p)} \mathcal{O}_\sigma(p').$$

We conclude that $\mathsf{Effect}(\sigma, \mathbf{m}_2, p) = \mathsf{Effect}(\sigma, \mathbf{m}_1, p) + \mathbf{m}_2(\mathsf{Sources}(\sigma, p)) - \mathbf{m}_1(\mathsf{Sources}(\sigma, p))$.

Induction step: Suppose that the hypothesis holds for all the sequences of multi-transfers where there are $i \leq k$ transfers $\langle P, p_t \rangle \in \mathsf{Transfer}(\sigma)$ such that $p_t = p$. Suppose that $\sigma = \sigma_1 \cdot \sigma_2$ is a sequences of multi-transfers such that there is k transfers $\langle P, p_t \rangle \in \mathsf{Transfer}(\sigma_1)$ with $p_t = p$ and there is one transfer $\langle P, p_t \rangle \in \mathsf{Transfer}(\sigma_2)$ with $p_t = p$. Moreover, suppose that

$$\mathbf{m}_1 \rightarrowtail^{\sigma_1} \mathbf{n}_1 \rightarrowtail^{\sigma_2} \mathbf{m}'_1$$

and

$$\mathbf{m}_2 \rightarrowtail^{\sigma_1} \mathbf{n}_2 \rightarrowtail^{\sigma_2} \mathbf{m}'_2.$$

By the induction hypothesis and the monotonicity property of MITN, we have

$$\mathsf{Effect}(\sigma_1, \mathbf{m}_2, p) = \mathsf{Effect}(\sigma_1, \mathbf{m}_1, p) + \mathbf{m}_2(\mathsf{Sources}(\sigma_1, p)) - \mathbf{m}_1(\mathsf{Sources}(\sigma_1, p)) \tag{5.1}$$

and

$$\mathsf{Effect}(\sigma_2, \mathbf{n}_2, p) = \mathsf{Effect}(\sigma_2, \mathbf{n}_1, p) + \mathbf{n}_2(\mathsf{Sources}(\sigma_2, p)) - \mathbf{n}_1(\mathsf{Sources}(\sigma_2, p)) \tag{5.2}$$

By definition, we have $\mathsf{Effect}(\sigma, \mathbf{m}_2, p) = \mathsf{Effect}(\sigma_1, \mathbf{m}_2, p) + \mathsf{Effect}(\sigma_2, \mathbf{n}_2, p)$ and $\mathsf{Effect}(\sigma, \mathbf{m}_1, p) = \mathsf{Effect}(\sigma_1, \mathbf{m}_1, p) + \mathsf{Effect}(\sigma_1, \mathbf{n}_2, p)$.

From (5.1) and (5.2), we obtain

$$\mathsf{Effect}(\sigma, \mathbf{m}_2, p) = \mathsf{Effect}(\sigma, \mathbf{m}_1, p) + \Delta$$

where $\Delta = \mathbf{m}_2(\mathsf{Sources}(\sigma_1, p)) + \mathbf{n}_2(\mathsf{Sources}(\sigma_2, p)) - (\mathbf{m}_1(\mathsf{Sources}(\sigma_1, p)) + \mathbf{n}_1(\mathsf{Sources}(\sigma_2, p))$. Let $\mathsf{Sources}(\sigma_2, p) = S_1 \cup S_2$ such that $S_1 \subseteq \mathsf{Sources}(\sigma_1, p)$ and $S_2 \cap \mathsf{Sources}(\sigma_1, p) = \emptyset$. So, S_1 is the set of places that are sources in σ_1 and S_2 is the set of places that are sources in σ_2 but not in σ_1 (notice that this set may be empty)

The number of tokens in S_1 after firing σ_1 is always the same (whatever the initial marking). Indeed, we can decompose σ_1 for each place $p' \in \mathsf{Sources}(\sigma_1, p)$ in a sequence $\sigma_{11} \cdot M \cdot \sigma_{12}$ where M contains a transfer $\langle P, p \rangle$ with $p' \in P$ and σ_{12} does not contain such a transfer to p. As the places in $\mathsf{Sources}(\sigma_1, p)$ cannot be the target of a transfer ((ii) of Definition 63) nor the source of Petri net arcs ((iii) of Definition 63), the number of tokens in p' after firing σ_1 is equal to the number of tokens added by Petri net arcs when firing $M \cdot \sigma_{12}$ and so $\mathbf{n}_2(S_1) = \mathbf{n}_1(S_1)$. Hence, we have that

$$\Delta = \mathbf{m}_2(\mathsf{Sources}(\sigma_1, p)) + \mathbf{n}_2(S_2) - (\mathbf{m}_1(\mathsf{Sources}(\sigma_1, p)) + \mathbf{n}_1(S_2))$$

Moreover, as the number of tokens added by σ_1 in the places in S_2 is constant (only Petri net arcs add tokens in those places), we have $\mathbf{n}_2(S_2) - \mathbf{n}_1(S_2) = \mathbf{m}_2(S_2) - \mathbf{m}_1(S_2)$, and hence

$$\Delta = \mathbf{m}_2(\mathsf{Sources}(\sigma_1, p)) + \mathbf{m}_2(S_2) - (\mathbf{m}_1(\mathsf{Sources}(\sigma_1, p)) + \mathbf{m}_1(S_2))$$

Finally, by definition of S_1 and S_2, we have that $\mathsf{Sources}(\sigma_1, p) \cup S_2 = \mathsf{Sources}(\sigma, p)$ and we obtain that

$$\Delta = \mathbf{m}_2(\mathsf{Sources}(\sigma, p)) - (\mathbf{m}_1(\mathsf{Sources}(\sigma, p))$$

and we conclude that

$$\mathsf{Effect}(\sigma, \mathbf{m}_2, p) = \mathsf{Effect}(\sigma, \mathbf{m}_1, p) + \mathbf{m}_2(\mathsf{Sources}(\sigma, p)) - (\mathbf{m}_1(\mathsf{Sources}(\sigma, p))$$

□

Assume a sequence σ of multi-transfers where there are transfers to the target place p. If, for instance, there are 5 tokens in the source places of the transfers to p in $\mathbf{m}_1$ and 7 tokens in the source places of the transfers to p in $\mathbf{m}_2$, then Lemma 21 states that σ will add two more tokens in p when fired from $\mathbf{m}_2$ instead of $\mathbf{m}_1$ (if σ is firable from $\mathbf{m}_1$ and $\mathbf{m}_2$).

Now, we are fully equipped to complete the proof sketched above.

Lemma 22 At each iteration of the main loop of Algorithm 21, the conditions C_1 and C_2 are fulfilled.

Proof. We already noticed that $(T_0, fixpoint_0)$ satisfies C_1 and C_2. It remains thus to show that if $(T_i, fixpoint_i)$, the tree and the set of ω-markings computed by Algorithm 21 at the end of the ith iteration of the main loop (line 1), satisfies the conditions C_1 and C_2 and the algorithm is not terminated, then $(T_{i+1}, fixpoint_{i+1})$ satisfies also those conditions, even if $\mathsf{AccIsolated}$ is applied. We reason by induction on the number of ω-markings added to $fixpoint$ during the $(i+1)th$ iteration of the main loop of Algorithm 21 (line 1).

Base case: $n = 0$. T_i and $fixpoint_i$ are not changed and so satisfy the properties C_1 and C_2.

Induction step: Suppose that the properties C_1 and C_2 are preserved after adding n ω-markings into fixpoint. Let us prove that those properties are still verified after adding one more ω-marking.

Two cases hold. Either we apply the acceleration function or not. In the last case, suppose that we have $\mathbf{m} \rightarrowtail^M \mathbf{m}'$ and that Algorithm 21 adds $\mathbf{m}'$ into $fixpoint$ and adds $\langle \mathbf{m}, \mathbf{m}' \rangle$ to T with the label M. If $\mathbf{m}$ is a reachable marking, so does $\mathbf{m}'$. Hence, we can associate the infinite sequence of reachable markings $\mathbf{m}'_1 \mathbf{m}'_2 \dots$ to $\mathbf{m}'$ such that $\mathbf{m}'_i = \mathbf{m}'$ for all $i \geq 1$. Moreover, we can associate the infinite sequence $\sigma_1 \sigma_2 \dots$ to the new arc $\langle \mathbf{m}, \mathbf{m}' \rangle$ in T where $\sigma_i = M$ for all $i \geq 1$. In that case, we have trivially $\mathbf{m}_i \rightarrowtail^{\sigma_i} \mathbf{m}'_i$ for all $i \geq 1$ (where $\mathbf{m}_1 \mathbf{m}_2 \dots$ is the infinite sequence of reachable markings associated to $\mathbf{m}$ by induction hypothesis with $\mathbf{m}_i = \mathbf{m}$ for all $i \geq 1$).

In the case where $\mathbf{m}$ is an $U - iwl$ of the infinite increasing sequence of reachable markings $\mathcal{S}^{\mathbf{m}} = \mathbf{m}_1 \mathbf{m}_2 \dots$, following Lemma 20, we can associate an infinite sequence of reachable markings $\mathcal{S} = \mathbf{m}'_1 \mathbf{m}'_2 \dots$ such that $\mathbf{m}'$ is an U-iwl of $\mathcal{S}$. Moreover, we can associate the infinite sequence $\sigma_1 \sigma_2 \dots$ to the arc $\langle \mathbf{m}, \mathbf{m}' \rangle$ with $\sigma_i = M$ for all $i \geq 1$ and such that there is $j \geq 0$ with $\mathbf{m}_{j+i} \rightarrowtail^M \mathbf{m}'_i$ for all $i \geq 1$ (notice that we can always suppress a prefix of size j in all the sequences $\mathcal{S}^{\mathbf{n}}$ considered up to now without endangering C_1 and C_2, so that the prefix $\mathbf{m}_1 \mathbf{m}_2 \dots \mathbf{m}_{j-1}$ can be removed).

In the case where the acceleration function is applied, we proceed as follows.

Suppose that Algorithm 21 applies $\mathsf{AccIsolated}$ on the ω-markings $\mathbf{m}$ and $\mathbf{m}'$ with $\mathbf{m} \prec \mathbf{m}'$, the set of multi-transfers $\mathcal{T}$ and the upward closed set U and suppose that $\mathbf{m}''$ is the result

$$
\begin{tikzcd}
\mathbf{n}_1(=\mathbf{m}) \arrow[d, dash] & \mathbf{n}_{11} \arrow[d, "\sigma_{11}"] & \preccurlyeq & \mathbf{n}_{12} \arrow[d, "\sigma_{12}"] & \preccurlyeq & \mathbf{n}_{13} \arrow[d, "\sigma_{13}"] & \cdots \\
\mathbf{n}_2 \arrow[d, dash] & \mathbf{n}_{21} \arrow[d, "\sigma_{21}"] & \preccurlyeq & \mathbf{n}_{22} \arrow[d, "\sigma_{22}"] & \preccurlyeq & \mathbf{n}_{23} \arrow[d, "\sigma_{23}"] & \cdots \\
\mathbf{n}_3 \arrow[d, dash] & \mathbf{n}_{31} \arrow[d, "\sigma_{31}"] & \preccurlyeq & \mathbf{n}_{32} \arrow[d, "\sigma_{32}"] & \preccurlyeq & \mathbf{n}_{33} \arrow[d, "\sigma_{33}"] & \cdots \\
\vdots & \vdots & & \vdots & & \vdots & \\
\mathbf{n}_k \arrow[d, "M"] & \mathbf{n}_{k1} \arrow[d, "M"] & \preccurlyeq & \mathbf{n}_{k2} \arrow[d, "M"] & \preccurlyeq & \mathbf{n}_{k3} \arrow[d, "M"] & \cdots \\
\mathbf{m}' & \mathbf{m}'_1 & \preccurlyeq & \mathbf{m}'_2 & \preccurlyeq & \mathbf{m}'_3 & \cdots
\end{tikzcd}
$$

Figure 5.12: Construction during the proof.

of AccIsolated. Let $\mathbf{n}_1 \ldots \mathbf{n}_k$ be such that $\mathbf{n}_1 = \mathbf{m}$, $\mathbf{m}'$ is constructed from $\mathbf{n}_k$ by firing the multi-transfer M and $\mathbf{n}_2, \ldots \mathbf{n}_{k-1}$ is the sequence of ω-markings corresponding to the path from the node labeled by $\mathbf{m}$ to the node labeled by $\mathbf{n}_k$ in the tree (see Figure 5.12).

For $i \geq 1$, we define $\sigma_i = \sigma_{i1} \cdot \ldots \cdot \sigma_{ik-1} \cdot M$ where σ_{ij} $(1 \leq j \leq k-1)$ is the ith sequence of multi-transfers associated to the edge from $\mathbf{n}_j$ to $\mathbf{n}_{j+1}$ by induction hypothesis on C_2 (see Figure 5.12: $\mathcal{S}^{\mathbf{n}_i} = \mathbf{n}_{i1}\mathbf{n}_{i2}\ldots$); again, from Lemma 20, we may fire M from the $\mathbf{n}_{ki}$'s, possibly after dropping some prefix of $\mathcal{S}^{\mathbf{n}_k}$ and from all the other $\mathcal{S}^{\mathbf{n}}$ already constructed. We then have that $\mathbf{m}'$ is an $U - iwl$ of $\mathbf{m}'_1\mathbf{m}'_2\ldots$. By construction, we have $\mathbf{m}_i = \mathbf{n}_{1i} \rightarrowtail^{\sigma_i} \mathbf{m}'_i$ for any $i \geq 1$, with $\mathcal{S}^{\mathbf{m}} = \mathbf{m}_1\mathbf{m}_2\ldots$ Notice that, by construction, σ_i is composed of multi-transfers from $\mathcal{T}$ and all the multi-transfers in $\mathcal{T}$ appear at least ones in σ_i (for $i \geq 1$).

We now construct the infinite sequence of ω-markings $\mathbf{m}''_1\mathbf{m}''_2\ldots$ as follows. For $i \geq 1$, $\mathbf{m}''_i$ is constructed by firing i times σ_i from some $\mathbf{m}'_{k_i}$, a reachable marking of the sequence $\mathcal{S}^{\mathbf{m}'} = \mathbf{m}'_1\mathbf{m}'_2\ldots$ associated to $\mathbf{m}'$. We shall show that such a marking $\mathbf{m}'_{k_i}$ exists. Let us first consider σ_1; since σ_1 is firable from $\mathbf{m}_1 = \mathbf{n}_{11}$ and $\mathbf{m}_1 \preccurlyeq \mathbf{m} \prec \mathbf{m}'$, σ_1 is also firable from $\mathbf{m}'$; if $\mathbf{m}'_1 = \mathbf{m}'$, we are done, otherwise applying Corollary 14, we may see that σ_1 is firable from some $\mathbf{m}'_{k_1}$ (and the following ones), yielding $\mathbf{m}''_1$.

Suppose that we have constructed $\mathbf{m}''_i$ for $i < j$ and let us construct $\mathbf{m}''_j$. As the sequence $\mathbf{m}''_1\mathbf{m}''_2\ldots$ must be strictly increasing, we choose $\mathbf{m}'_{k_j}$, the reachable marking from which $\mathbf{m}''_j$ is constructed by firing $(\sigma_j)^j$, as follows. In the following, we will consider $\mathbf{l} = lub(\{\mathbf{m}_1, \mathbf{m}_2, \ldots\})$ and $\mathbf{l}' = lub(\{\mathbf{m}'_1, \mathbf{m}'_2, \ldots\})$. Moreover, we consider in the following that all the infinite sequences $\mathcal{S}^{\mathbf{n}_j} = \mathbf{n}_{j1}\mathbf{n}_{j2}\ldots$ with $(j = 1 \ldots k+1$, with $\mathbf{n}_{k+1} = \mathbf{m}')$ are such that for each place p, either $\mathbf{n}_{ji}(p) = \mathbf{n}_{j(i+1)}(p)$ for all $i \geq 1$ or $\mathbf{n}_{ji}(p) < \mathbf{n}_{j(i+1)}(p)$ for all $i \geq 1$, which is always possible by dropping some common sub-sequences.

1. For all the places p that are neither a source nor a target of a transfer in $\mathsf{Transfer}(\mathcal{T})$.
   - If $\mathbf{m}(p) = \mathbf{m}'(p)$, $\mathbf{m}''(p) = \mathbf{m}'(p)$ and we consider two cases.
     (a) $\mathbf{m}''(p) = +\infty$. In this case, either

i. $\mathbf{l}'(p) = +\infty$. Let k be the number of tokens removed from p by Petri net arcs when firing σ_j. We choose $\mathbf{m}'_{k_j}$ such that $\mathbf{m}'_{k_j}(p) > \mathbf{m}''_{j-1}(p) + j \cdot k$;

ii. or $\mathbf{l}'(p) \in \mathbb{N}$. In that case, since $\mathbf{m}'$ is an $U-iwl$ of $\mathbf{m}'_1\mathbf{m}'_2\ldots$, p is a source of at least one multi-transfer in the MITNand since $+\infty = \mathbf{m}'(p) > \mathbf{l}'(p)$ we have that $\mathbf{l}'(p) \geq \mathbf{m}_U(p)$. Hence no Petri net arcs can remove tokens from p (condition (iii) if Definition 63), and we conclude that the effect of firing σ_j is not negative on p. The same is true for any σ_k, so that $\mathbf{m}'_k(p) \geq \mathbf{m}_k(p)$ for all $k \geq 1$. We consider two cases.

A. Either $\mathbf{l}(p) = \mathbf{l}'(p)$. In that case, σ_l has no effect on p for all $l \geq 1$. For all $i, l \geq 1$, if we have $\mathbf{m}'_i \rightarrowtail^{\sigma_l^l} \mathbf{n}$, we are guaranteed to have $\mathbf{n}(p) = \mathbf{l}'(p)$ with $\mathbf{n}(p) \geq \mathbf{m}_U(p)$. In this case no further assumption is needed for $\mathbf{m}'_{k_j}$;

B. or $\mathbf{l}(p) < \mathbf{l}'(p)$. In that case, all the σ_l add the same (constant) number of tokens in p since $\mathbf{l}'(p) \in \mathbb{N}$. Let $k = \mathbf{l}'(p) - \mathbf{l}(p)$. Firing σ_l $(l \geq 1)$ adds k tokens in p. For all $i, l \geq 1$, if we have $\mathbf{m}'_i \rightarrowtail^{\sigma_l^l} \mathbf{n}$, then we have $\mathbf{n}(p) = \mathbf{m}'_i(p) + l \cdot k$. We conclude that for all $i \geq 1$ if we have $\mathbf{m}'_i \rightarrowtail^{\sigma_j^j} \mathbf{n}$, we are guaranteed to have $\mathbf{n}(p) > \mathbf{m}''_{j-1}(p)$ and $\mathbf{n}(p) \geq \mathbf{m}_U(p)$. In this case no further assumption is needed for $\mathbf{m}'_{k_j}$.

(b) $\mathbf{m}''(p) \in \mathbb{N}$. In this case, $\mathbf{m}(p) = \mathbf{m}_i(p) = \mathbf{m}'_i(p) = \mathbf{m}'(p) = \mathbf{m}''(p)$ for all $i \geq 1$ and σ_j as well as the other ones has no effect on the number of tokens in p and we have $\mathbf{m}'_i(p) = \mathbf{m}'_{i+1}(p)$ for all $i \geq 1$. We conclude that for all $i \geq 1$, if we have $\mathbf{m}'_i \rightarrowtail^{\sigma_j^j} \mathbf{n}$, then $\mathbf{n}(p) = \mathbf{m}'(p) = \mathbf{m}''_{j-1}(p)$. In this case no further assumption is needed for $\mathbf{m}'_{k_j}$.

- If $\mathbf{m}(p) < \mathbf{m}'(p)$, we consider two cases.

(a) $\mathbf{m}'(p) = +\infty$. The following cases hold.

i. $\mathbf{l}'(p) = +\infty$. Let k be the number of tokens removed from p by Petri nets arcs when σ_j is fired. We choose $\mathbf{m}'_{k_j}$ such that $\mathbf{m}'_{k_j} > \mathbf{m}''_{j-1} + j \cdot k$.

ii. $\mathbf{l}'(p) \in \mathbb{N}$. By definition of U-iwl, we know that $\mathbf{l}'(p) \geq \mathbf{m}_U(p)$. p is source of transfers in some multi-transfers that are not used when firing σ_i $(i \geq 1)$. As no Petri net arcs can remove tokens from p, we conclude that the effect of firing σ_j is not negative on p. We consider two cases.

A. Either $\mathbf{l}(p) = \mathbf{l}'(p)$. For all $i, l \geq 1$, if we have $\mathbf{m}'_i \rightarrowtail^{\sigma_l^l} \mathbf{n}$, we are guaranteed to have $\mathbf{n}(p) = \mathbf{l}'(p)$ with $\mathbf{n}(p) \geq \mathbf{m}_U(p)$. In this case no further assumption is needed for $\mathbf{m}'_{k_j}$;

B. or $\mathbf{l}(p) < \mathbf{l}'(p)$. Let $k = \mathbf{l}'(p) - \mathbf{l}(p)$. Firing σ_i $(i \geq 1)$ adds k tokens in p. For all $i, l \geq 1$, if we have $\mathbf{m}'_i \rightarrowtail^{\sigma_l^l} \mathbf{n}$, then we have $\mathbf{n}(p) = \mathbf{m}'_i(p) + l \cdot k$. We conclude that for all $i \geq 1$ if we have $\mathbf{m}'_i \rightarrowtail^{\sigma_j^j} \mathbf{n}$, we are guaranteed to have $\mathbf{n}(p) > \mathbf{m}''_{j-1}(p)$ and $\mathbf{n}(p) \geq \mathbf{m}_U(p)$. In this case no further assumption is needed for $\mathbf{m}'_{k_j}$.

(b) $\mathbf{m}'(p) \in \mathbb{N}$. Then, $\mathbf{m}_i(p) = \mathbf{m}(p)$ and $\mathbf{m}'_i(p) = \mathbf{m}'(p)$ for all $i \geq 1$. Let $k = \mathbf{m}'(p) - \mathbf{m}(p) > 0$. Firing σ_i $(i \geq 1)$ adds k tokens in p and p cannot be an obstacle to fire σ_i again. For all $i, l \geq 1$, if we have $\mathbf{m}'_i \rightarrowtail^{\sigma_l^l} \mathbf{n}$, then we have $\mathbf{n}(p) = \mathbf{m}'_i(p) + l \cdot k$. We conclude that for all $i \geq 1$ if we have $\mathbf{m}'_i \rightarrowtail^{\sigma_j^j} \mathbf{n}$, we are guaranteed to have $\mathbf{n}(p) > \mathbf{m}''_{j-1}(p)$. In this case no further assumption is needed for $\mathbf{m}'_{k_j}$.

2. For all the places p that are source of a transfer (hence never a target of a transfer, nor a source of a Petri net arc, and the target is unique (Conditions (i)-(ii)-(iii) of Definition 63)) in $\mathsf{Transfer}(\mathcal{T})$, p may not be an obstacle to replay any σ_k any number of time from any marking. We can decompose σ_j (but the same is true for each σ_i with $i < j$) such that $\sigma_j = \sigma_1 \cdot M \cdot \sigma_2$ where M contains a transfer $\langle P, p_t \rangle$ with $p \in P$ and σ_2 does not contain such a transfer. Since the transfers of M empty p, the number of tokens in p after firing σ_j^k, whatever $k \geq 1$, from any marking is equal to the number of tokens added into p by Petri nets arcs when firing $M \cdot \sigma_2$. We consider the following cases.

   (a) $\mathbf{m}''(p) = +\infty$. Two cases holds.

      i. $\mathbf{l}'(p) = +\infty$. Then $\mathbf{m}'_1\mathbf{m}'_2$ is strictly increasing on p and it follows that the number of tokens in p after firing σ_j is greater than after firing σ_{j-1}. We conclude that for all $i \geq 1$, if we have $\mathbf{m}'_i \xrightarrow{\sigma_j^j} \mathbf{n}$, then $\mathbf{n}(p) = \mathbf{m}'_j(p) > \mathbf{m}''_{j-1}(p) = \mathbf{m}'_{j-1}(p)$. In this case no further assumption is needed for $\mathbf{m}'_{k_j}$.

      ii. $\mathbf{l}'(p) \in \mathbb{N}$, so that $\mathbf{l}'(p) = \mathbf{m}'_i(p)$ for all $i \geq 1$. We consider two cases.

         A. $\mathbf{m}'(p) = +\infty$. We have $\mathbf{l}'(p) \geq \mathbf{m}_U(p)$ and we conclude that if we have $\mathbf{m}'_i \xrightarrow{\sigma_j^j} \mathbf{n}$ for some $i \geq 1$, then $\mathbf{n}(p) = \mathbf{m}'_j(p) = \mathbf{l}'(p) = \mathbf{m}''_{j-1}$. In this case no further assumption is needed for $\mathbf{m}'_{k_j}$.

         B. $\mathbf{m}'(p) \in \mathbb{N}$. In that case, $\mathsf{AccIsolated}$ puts $+\infty$ into p and we have $\mathbf{l}'(p) = \mathbf{m}'(p) \geq \mathbf{m}_U(p)$. We conclude that for all $i \geq 1$, if we have $\mathbf{m}'_i \xrightarrow{\sigma_j^j} \mathbf{n}$, then $\mathbf{n}(p) = \mathbf{m}'_j(p) = \mathbf{l}'(p) = \mathbf{m}'_{j-1}(p)$ with $\mathbf{n}(p) \geq \mathbf{m}_U(p)$. In this case no further assumption is needed for $\mathbf{m}'_{k_j}$.

   (b) $\mathbf{m}''(p) \in \mathbb{N}$ and $\mathbf{m}''(p) = \mathbf{m}'(p) = \mathbf{m}(p)$. In this case if we have $\mathbf{m}'_i \xrightarrow{\sigma_j^j} \mathbf{n}$, then we are guaranteed to have $\mathbf{n}(p) = \mathbf{m}'(p)$, for all $i \geq 1$. In this case no further assumption is needed for $\mathbf{m}'_{k_j}$.

3. For all the places p that are the target of a transfer in $\mathsf{Transfer}(\mathcal{T})$. We have that p is not a source of a transfer (Condition (ii) of Definition 63) and since $\mathbf{m}'$ is an $U - iwl$ of $\mathbf{m}'_1\mathbf{m}'_2\ldots$ we have $\mathbf{m}'(p) = \mathbf{l}'(p)$. Let k be the number of tokens removed from p by Petri net arcs when firing σ_j. We consider two cases.

   - $\mathbf{m}(p) = \mathbf{m}'(p)$. We consider two cases.

     (a) $\mathbf{m}''(p) = +\infty$. We consider two cases.

        i. $\mathbf{m}'(p) = \mathbf{l}'(p) = +\infty$. We choose $\mathbf{m}'_{k_j}$ such that $\mathbf{m}'_{k_j}(p) > \mathbf{m}''_{j-1}(p) + j \cdot k$.

        ii. $\mathbf{m}'(p) \in \mathbb{N}$, so that $\mathbf{m}'(p) = \mathbf{l}'(p) = \mathbf{m}'_i(p)$ for all $i \geq 1$. This corresponds to the case where $\mathsf{AccIsolated}$ puts $+\infty$ into p and there exists $p' \in \mathsf{Source}(\sigma_j, p)$[7] such that $\mathbf{m}(p') < \mathbf{m}'(p')$. We also have $\mathbf{l}(\mathsf{Source}(\sigma_j, p)) \in \mathbb{N}$ and $\mathsf{Effect}(\mathbf{m}_k, \sigma_k, p) = 0$ for all $k \geq 1$ since otherwise we would have $\mathbf{m}'(p) = \mathbf{l}'(p) = +\infty$. We consider two cases.

[7] Remember that the σ_l $(l \geq 1)$ is constructed from the whole set $\mathcal{T}$ of multi-transfers so that $\mathsf{Sources}(\sigma_i, p) = \mathsf{Sources}(\sigma_l, p)$ for all $i, l \geq 1$.

A. Either there exists $p'' \in \mathsf{Source}(\sigma_j, p)$ with $\mathbf{l}'(p'') = +\infty$. Following Lemma 21, we choose in that case $\mathbf{m}'_{k_j}$ such that $\mathbf{m}'_{k_j}(\mathsf{Source}(\sigma_j, p)) - \mathbf{m}_j(\mathsf{Source}(\sigma_j, p)) > \mathbf{m}''_{j-1}(p) + j \cdot k$;

B. or there does not exist $p' \in \mathsf{Source}(\sigma_j, p)$ with $\mathbf{l}'(p') = +\infty$. In that case, we have $\mathbf{m}'(\mathsf{Source}(\sigma_j, p)) \in \mathbb{N}$. Let $k' = \sum_{p' \in \mathsf{Sources}(\sigma_j, p)}(\mathbf{m}'(p') - \mathbf{m}(p'))$. As $\mathsf{AccIsolated}$ puts $+\infty$ in p, we have $k' > 0$. Following Lemma 21, we have that if $\mathbf{m}'_i \rightarrowtail^{\sigma_l^l} \mathbf{n}$, then $\mathbf{n}(p) = \mathbf{m}'_i(p) + l \cdot k'$, for all $i, l \geq 1$. We conclude that for all $i \geq 1$ if $\mathbf{m}_i \rightarrowtail^{\sigma_j^j} \mathbf{n}$, then $\mathbf{n}(p) > \mathbf{m}''_{j-1}(p)$. So, in this case no further assumption is needed for $\mathbf{m}'_{k_j}$.

(b) $\mathbf{m}''(p) \in \mathbb{N}$. In this case $\mathbf{m}''(p) = \mathbf{m}'(p)$ and there does not exist $p' \in \mathsf{Source}(\sigma_j, p)$ such that $\mathbf{m}(p') < \mathbf{m}'(p')$. Following Lemma 21, we conclude that if we have $\mathbf{m}'_i \rightarrowtail^{\sigma_l^l} \mathbf{n}$, then $\mathbf{n}(p) = \mathbf{m}'(p)$, for all $i, l \geq 1$, so that $\mathbf{m}''_j(p) = \mathbf{m}''_{j-1}(p)$. In this case no further assumption is needed for $\mathbf{m}'_{k_j}$.

- $\mathbf{m}(p) < \mathbf{m}'(p)$. Two cases holds.

(a) If $\mathbf{l}'(p) = +\infty$, we choose $\mathbf{m}'_{k_j}$ such that $\mathbf{m}'_{k_j}(p) > \mathbf{m}''_{j-1}(p) + j \cdot k$.

(b) If $\mathbf{l}'(p) \in \mathbb{N}$, then $\mathsf{AccIsolated}$ puts $+\infty$ into p. We consider two cases.

i. There exists $p' \in \mathsf{Source}(\sigma_j, p)$ such that $\mathbf{l}'(p') = +\infty$. Following Lemma 21, we choose $\mathbf{m}'_{k_j}$ such that $\mathbf{m}'_{k_j}(\mathsf{Source}(\sigma_j, p)) - \mathbf{m}_j(\mathsf{Source}(\sigma_j, p)) > \mathbf{m}''_{j-1}(p) + j \cdot k$.

ii. There does not exist $p' \in \mathsf{Source}(\sigma_j, p)$ such that $\mathbf{l}'(p') = +\infty$. In that case we have $\mathbf{m}'(\mathsf{Source}(\sigma_j, p)) \in \mathbb{N}$ and $\mathbf{m}(\mathsf{Source}(\sigma_j, p)) \in \mathbb{N}$. Let $k' = \sum_{p' \in \mathsf{Source}(\sigma_j, p)} \mathbf{m}'(p') - \mathbf{m}(p')$ and $k'' = \mathbf{m}'(p) - \mathbf{m}(p)$. Notice that $k' + k'' > 0$ since $\mathbf{m}'(p) > \mathbf{m}(p)$ and $\mathbf{m} \prec \mathbf{m}'$. Following Lemma 21, if we have $\mathbf{m}'_i \rightarrowtail^{\sigma_l^l} \mathbf{n}$, then $\mathbf{n}(p) = \mathbf{m}'_i(p) + l \cdot (k' + k'')$, for all $i, l \geq 1$. We conclude that if $\mathbf{m}_i \rightarrowtail^{\sigma_j^j} \mathbf{n}$, then $\mathbf{n}(p) > \mathbf{m}''_{j-1}(p)$, for all $i \geq 1$. So, in this case no further assumption is needed for $\mathbf{m}'_{k_j}$.

Notice that all the markings $\mathbf{m}''_i$ are reachable markings from S_0 since they are computed from reachable markings by firing sequences of multi-transfers. We conclude that $\mathbf{m}''$ is an $U - iwl$ of the strictly increasing sequence of reachable markings $\mathcal{S}^{\mathbf{m}''} = \mathbf{m}''_1 \mathbf{m}''_2 \ldots$. Indeed, for each place p such that $\mathbf{m}''(p) \in \mathbb{N}$, the sequence $\mathcal{S}^{\mathbf{m}''}$ is constant on p and $\mathbf{m}''(p) = \mathbf{m}''_i(p)$ for all $i \geq 1$. Moreover, for each place p such that $\mathbf{m}''(p) = +\infty$, either $\mathcal{S}^{\mathbf{m}''}$ is strictly increasing on p and hence there exists $i \geq 1$ such that for all $i' \geq i$ we have $\mathbf{m}''_{i'}(p) \geq \mathbf{m}_U(p)$, or p is a source place of a transfer to the target place p' such that $\mathcal{S}^{\mathbf{m}''}$ is constant on p with $\mathbf{m}''_i(p) \geq \mathbf{m}_U(p)$ for all $i \geq 1$, and strictly increasing on p'.

Moreover, the sequences $\sigma_1, \sigma_2^2, \sigma_3^3, \ldots$ contain by construction all the multi-transfers associated to the edge $\langle \mathbf{m}', \mathbf{m}'' \rangle$. Finally, let $I = \{k_1, k_2, \ldots\}$. After removing from all the other sequences of markings $\mathcal{S}^{\mathbf{n}}$ and all the other sequences of multi-transfers $\sigma_1 \sigma_2 \ldots$ considered up to now the elements of index i with $i \notin I$, we conclude that T and $fixpoint$ satisfy the two conditions C_1 and C_2.

We conclude that $(T_{n+1}, fixpoint_{n+1})$ satisfies the properties C_1 and C_2. □

From Lemma 22, we obtain the following result.

Lemma 23 Given a MITN $\mathcal{M}$ with the set of initial markings S_0 and an upward closed set U described by the minimal marking $\mathbf{m}_U$, Algorithm 21 returns *false*, if U is not reachable from S_0 in $\mathcal{M}$.

Proof. From Lemma 22, we obtain that if the algorithm terminates after k iterations, then $fixpoint_k$ contains ω-markings that are either reachable markings from S_0 or U-iwls, and hence U-weak limit. It follows that there exists a weak coverability set $\texttt{WCS}(\mathcal{M}, S_0)$ such that $dcones(fixpoint_k) \subseteq dcones(\texttt{WCS}(\mathcal{M}, S_0))$. We conclude that if U is not reachable, from Proposition 20 there is no $\mathbf{m} \in dcones(fixpoint_k)$ such that $\mathbf{m} \in U$ and Algorithm 21 returns $false$. □

Finally, we obtain the following theorem

Theorem 28 Algorithm 21 terminates and returns *true* if U is reachable from S_0, $false$ otherwise.

Proof. Termination is ensured by Lemma 18. That Algorithm 21 answers *true* when U is reachable from S_0 is ensured by Lemma 19. Finally, that Algorithm 21 answers $false$ when U is not reachable from S_0 is ensured by Lemma 23. □

5.4 Heuristics to Speed-up the Backward Approach

5.4.1 Structural Heuristics

In the backward reachability approach every place of a Petri net is considered as *unbounded.* In many practical cases, however, it may happen that some places of the net are bounded for any possible value of the parameters in the initial configuration. The *Structural Theory* of Petri nets [STC98] can help us to distinguish between bounded and unbounded places and, thus, to restrict the search space we have to explore backward. Let us recall the main results of this theory.

Let N be a Petri net with n places, m transitions, and the *token flow matrix* $\mathbf{C}$. We recall that $\mathbf{C}$ describes how tokens are reallocated by transitions: rows correspond to places, columns to transitions, and an element c_{ij} to the difference between the number of tokens added and removed for place i by the j-th transition. In the following, we will use $\cdot$ to denote the vector product

$$\mathbf{a}^T \cdot \mathbf{b} = a_1 b_1 + \ldots + a_n b_n$$

where $\mathbf{a}^T$ indicates the transpose of vector $\mathbf{a}$.

Place invariants [STC98] are among the possible information we can compute via a static analysis of N. A place invariant (also called P-semiflow) is a vector $\mathbf{p} = \langle p_1, \ldots, p_n \rangle$ with non-negative components that is a solution of the linear system

$$\mathbf{x}^T \ \cdot \ \mathbf{C} \ = \ \mathbf{0}, \ \mathbf{x} \in \mathbb{N}$$

where $\mathbf{x}$ is a vector of variables of dimension n and $\mathbf{0}$ is the vector were all the components are equal to 0. Given an initial marking $\mathbf{m}_0$, and a place invariant $\mathbf{p}$, the set

$$\mathcal{O}(\mathbf{m}_0, \mathbf{p}) = \{\mathbf{m} \mid \mathbf{p}^T \cdot \mathbf{m} = \mathbf{p}^T \cdot \mathbf{m}_0\}$$

over-approximates the reachability set of the Petri Net. This property follows from the definition of a place invariant, and from the *state equation*

$$\mathbf{m} = \mathbf{m}_0 + \mathbf{C} \cdot \sigma$$

that characterizes a generic marking $\mathbf{m}$ reachable from $\mathbf{m}_0$ via the sequence of transitions represented by the *firing vector* σ (see [STC98]): if $\sigma = \langle c_1, \ldots, c_m \rangle$ then the j-th transition has been fired c_j times to reach $\mathbf{m}$ from $\mathbf{m}_0$. As a consequence, the equation

$$\mathbf{p}^T \cdot \mathbf{x} = \mathbf{p}^T \cdot \mathbf{m_0}$$

valid for any place invariant $\mathbf{p}$ gives us a *structural invariant* we can use to analyze the net. Let us consider the example of Figure 5.3. The state equation is given as follows

$$\begin{pmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \end{pmatrix} = \begin{pmatrix} K \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} -1 & 1 & 1 & -1 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix}$$

where $m_1, \ldots, m_5$ and $c_1, \ldots, c_5$ are required to be non-negative. Place invariants can be computed then by solving the following linear system

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \end{pmatrix} \cdot \begin{pmatrix} -1 & 1 & 1 & -1 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 \end{pmatrix}$$

where $x_1, \ldots, x_5$ are also required to be non-negative. After simplifications, we get the following linear system

$$\begin{aligned} x_1 + x_2 &= x_5 \\ x_1 + x_3 &= x_4 \\ x_1, x_2, x_3, x_4, x_5 &\geq 0 \end{aligned}$$

This gives us, for instance, the following *place invariants*: $\mathbf{p_1} = \langle 0, 1, 0, 0, 1 \rangle$, $\mathbf{p_2} = \langle 0, 0, 1, 1, 0 \rangle$, and $\mathbf{p_3} = \langle 1, 0, 0, 1, 1 \rangle$. Actually, these place invariants are the minimal solutions (different from $\mathbf{0}$) that represent a basis to generate (via linear combinations) all other place invariants. Algorithms to compute generators of place invariants are mentioned, e.g., in [STC98]. When applied to the state equation with the parametric initial marking, we then obtain the following structural invariants:

$$\begin{aligned} x_2 + x_5 &= 1 \\ x_3 + x_4 &= 1 \\ x_1 + x_4 + x_5 &= K. \end{aligned}$$

Notice that the first two invariants, namely $x_2 + x_5 = 1$ and $x_3 + x_4 = 1$, imply that the places x_2, x_3, x_4 and x_5 always contain at most one token, i.e., they imply $x_i \leq 1$ for $i : 2, \ldots, 5$. Unfortunately, this information is not sufficient to prove our mutual exclusion property expressed via the constraint $x_4 + x_5 \leq 1$: a configuration in which both x_4 and

x_5 have one token satisfies the structural invariants but not the mutual exclusion property. The weakness of the invariants comes from the fact that they do not take into account the guards of the transitions, an information that is not represented in the matrix $\mathbf{C}$. Despite this problem, we can still try to exploit the information coming from the first two invariants.

A possible way to use the structural analysis would be to make what is usually called a *program specialization*, i.e., we could replace the subnet involving places linked by structural invariants (e.g. p_2, p_5 for $x_2+x_5 = 1$) with a control part (a finite-state net where each place corresponds to a configuration satisfying the invariants). Applying this approach, however, we face the risk to obtain nets of exponential size. As an alternative, we propose to use the structural invariants directly within our k-ucCSTs-based search scheme, as a heuristics to *prune the search space.*

Pruning the Backward Search Space

Let U be an upward-closed set of markings denoting *unsafe states*, and let $U' = U \cap_{i=1\ldots n} \mathcal{O}(\mathbf{m}_0, \mathbf{p}_i)$ where $\mathbf{p}_1, \ldots, \mathbf{p}_n$ are place invariants. We first note that if $U' = \emptyset$, then we can immediately infer that the net is *safe* with respect to the violations represented via U. As mentioned in the previous section, invariants are often not sufficient to directly verify the property, but they can be used to prune the backward search space. Let us consider again our example and the backward reachability graph of Figure 5.13. After the first iteration, we obtain the generators $\langle 1,1,1,0,1\rangle$ and $\langle 1,1,1,1,0\rangle$. They are not subsumed by pre-existing generators. The first generator defines a set of markings that has no intersection with the set of markings defined by the invariant $x_2 + x_5 = 1$, while the second generator defines a set of markings that have no intersection with the set of markings defined by the invariant $x_3 + x_4 = 1$. So we know that no markings defined by those two generators will be reached from any instance of the parametric initial marking (the invariants do not depend on the value of K). In fact, the set of markings satisfying the invariant over-approximates the set of reachable markings. As a consequence, we can stop the backward search after the first iteration.

$$\begin{tikzcd}
\text{Unsafe States} & \langle 0,0,0,1,1\rangle & \langle 0,0,0,2,0\rangle & \langle 0,0,0,0,2\rangle \\
\text{Iteration 1} & \langle 1,1,1,0,1\rangle \arrow[u, "t_1"'] & \langle 1,1,1,1,0\rangle \arrow[u, "t_1"'] & \\
 & \text{Violate Inv } x_2 + x_5 = 1 & \text{Violate Inv } x_3 + x_4 = 1 &
\end{tikzcd}$$

Figure 5.13: Backward search space pruned using structural invariants.

Let us now study how we can incorporate this idea in our k-ucCST-based backward search. We first note that, in general, the intersection between an upward-closed set U and the set of markings $\mathcal{O}(\mathbf{m}_0, \mathbf{p})$ induced by a place invariants $\mathbf{p}$ is not upward-closed. Thus, even if we work with a symbolic representation of invariants using linear constraints (e.g. $x_2 + x_5 = 1 \wedge x_3 + x_4 = 1$), we cannot conjunct the invariant with the intermediate results of the algorithm for pre^* and represent the result with a k-ucCST. For instance, the constraint $x_2 + x_5 = 1 \wedge x_2 \geq 1 \wedge x_5 \geq 0$ is equivalent to $x_2 = 1 \wedge x_5 = 0$, whose denotation is not upward-closed.

What we can do, however, is to remove, from the k-ucCSTs computed after each step of the symbolic reachability algorithm, *all the generators* that define sets of markings that have no intersection with the set $\mathcal{O}(\mathbf{m}_0, \mathbf{p})$. In fact, we have the following property.

Theorem 29 Given a Petri Net N with initial parametric marking $\mathbf{m}_0$, a place invariant $\mathbf{p}$, and an upward-closed set of markings U represented by a k-ucCST $\mathbf{S}$, suppose $cones(\mathbf{m}) \cap \mathcal{O}(\mathbf{m}_0, \mathbf{p}) = \emptyset$ for some $\mathbf{m} \in elem(\mathbf{S})$. Furthermore, let $\mathbf{S}'$ be the k-ucCST such that $elem(\mathbf{S}') = elem(\mathbf{S}) \setminus \{\mathbf{m}\}$, and $\mathbf{m}'_0$ be any instance of $\mathbf{m}_0$. Then,

$$\text{If } \mathbf{m}'_0 \in pre^*(cones(\mathbf{S})) \text{ then } \mathbf{m}'_0 \in pre^*(cones(\mathbf{S}')).$$

Proof. Per absurdum, suppose there exists an instance $\mathbf{m}'_0$ of $\mathbf{m}_0$ with $\mathbf{m}'_0 \in pre^*(cones(\mathbf{S}))$ and $\mathbf{m}'_0 \notin pre^*(cones(\mathbf{S}'))$. This implies that from $\mathbf{m}'_0$, hence also from $\mathbf{m}_0$, we can reach a marking $\mathbf{m}'$ generated by $\mathbf{m}$. But this contradict the hypothesis that $\mathbf{p}$ is a place invariant of N with $cones(\mathbf{m}) \cap \mathcal{O}(\mathbf{m}_0, \mathbf{p}) = \emptyset$. □

Thus, during the backward search, we can prune the search space by removing *all* generators $\mathbf{m}$, such that $cone(\mathbf{m})$ has an empty intersection with the set of markings defined by a structural invariant. We call such generators *useless*. As shown in the following theorem, this process can be done in polynomial time in the size of the k-ucCSTs.

Theorem 30 Given a Petri Net N with initial parametric marking $\mathbf{m}_0$, a k-ucCST $\mathbf{S}$ and a place invariant $\mathbf{p}$, such that $M = \mathbf{p}^T \cdot \mathbf{m}_0$ is an integer value (no parameter occurs in M). Then, there exists an algorithm to compute the k-ucCST $\mathbf{S}'$ such that $\mathbf{m} \in elem(\mathbf{S}')$ if and only if $\mathbf{m} \in elem(\mathbf{S})$ and $cone(\mathbf{m}) \cap \mathcal{O}(\mathbf{m}_0, \mathbf{p}) \neq \emptyset$, with a complexity polynomial in the size of $\mathbf{S}$, $\mathbf{p}$, and in the value M.

Proof. Let $\mathbf{p} = \langle p_1, \ldots, p_k \rangle$ and $M = \mathbf{p}^T \cdot \mathbf{m}_0$.

The algorithm works as follows. We first constructs (in quadratic time) a pre-sharing tree $\mathbf{T}$ such that $\mathbf{m} \in elem(\mathbf{T})$ if and only if $\mathbf{m} \in elem(\mathbf{S})$ and $cone(\mathbf{m}) \cap \mathcal{O}(\mathbf{m}_0, \mathbf{p}) \neq \emptyset$.

$\mathbf{T}$ is defined as follows. Its set of nodes $N^{\mathbf{T}}$ consists of $root^{\mathbf{T}}$, $end^{\mathbf{T}}$, and the nodes of the i-th layer (in $N_i^{\mathbf{T}}$) are the pairs $\langle s, v \rangle$ composed of a node $s \in N_i^{\mathbf{S}}$, and a value $v \in \{0, \ldots, M\}$. We set $val(\langle s, v \rangle) = val(s)$. The successor relation $succ^{\mathbf{T}}$ is defined as (recall that $\mathbf{p} = \langle p_1, \ldots, p_k \rangle$):

- $succ^{\mathbf{T}}(root^{\mathbf{T}})$ contains all nodes $\langle s, v \rangle$ such that $s \in succ^{\mathbf{S}}(root^{\mathbf{S}})$ and $v = p_1 \cdot val(s)$;
- for all the nodes $\langle s, v \rangle$ of the ℓ-th layer ($\ell < k$), $succ^{\mathbf{T}}(\langle s, v \rangle) = \{ \langle s', v' \rangle$ such that $s' \in succ^{\mathbf{S}}(s)$ and $v' = v + p_{\ell+1} \cdot val(s') < M \}$. Notice that if $p_i = 0$ then $\langle s', v \rangle \in succ^{\mathbf{T}}(\langle s, v \rangle)$ if and only if $s' \in succ^{\mathbf{S}}(s)$;
- for all nodes $\langle s, v \rangle$ of the k-th layer $succ^{\mathbf{T}}(\langle s, v \rangle) = \{ end^{\mathbf{T}} \}$.

Notice that the nodes without successors (except the ending node) must be removed from $\mathbf{T}$. The resulting pre-sharing tree has the following property: If $s' \in succ^{\mathbf{S}}(s)$ then for each value $v \in \{0, \ldots, M\}$, there is *at most one* v' such that $\langle s, v' \rangle \in succ^{\mathbf{T}}(\langle s, v \rangle)$. Furthermore, $\mathbf{t} \in elem(\mathbf{T})$ if and only if $\mathbf{t} \in elem(\mathbf{S})$ and $\mathbf{p}^T \cdot \mathbf{t} = \mathbf{p}^T \cdot \mathbf{m}_0$. The construction of $\mathbf{T}$ is done in time *polynomial* in the size of $\mathbf{S}$, M and $\mathbf{p}$. Finally, to turn $\mathbf{T}$ into an k-ucCST we have to enforce the maximal sharing of suffixes in $\mathbf{T}$. This can also be done in *polynomial* time using the algorithm Figure 28 in Appendix .1. □

The algorithm presented in the proof of Theorem 30 is used to prune k-ucCSTs with respect to one place invariant during the backward search. If we have several place invariants, then this algorithm is applied for each of them.

As we only need $\lceil log_2(M) \rceil$ bits to encode a value M, the algorithm presented in the proof of Theorem 30 is only a *pseudo* polynomial algorithm, i.e. it is polynomial in M but exponential in its encoding. In the next section, we present a polynomial heuristics to remove useless generators.

Approximation of the Structural Heuristic

In the previous section we presented a pseudo polynomial algorithm to remove all the *useless* paths of a k-ucCST according to a structural invariant. We present now an approximated algorithm that can be used as a heuristics to remove *useless* paths. The new algorithm is more efficient in practical cases. To describe the heuristics, we need the following definitions. Let $\mathbf{S}$ be a k-ucCST , and let $e = (\mathbf{v}, \mathbf{w})$ be an edge of $\mathbf{S}$ connecting nodes of two adjacent layers. We define $elem_e(\mathbf{S})$ as the set of tuples from $elem(\mathbf{S})$ denoted by paths of $\mathbf{S}$ passing through e. Formally, $\mathbf{m} = \langle val(n_1), \ldots, val(n_i), val(\mathbf{v}), val(\mathbf{w}), val(n_{i+1}), \ldots, val(n_k)\rangle \in elem_e(\mathbf{S})$ if and only if there exists a path $\langle root^{\mathbf{S}}, n_1, \ldots, n_i, \mathbf{v}, \mathbf{w}, n_{i+1}, \ldots, v_k, end^{\mathbf{S}}\rangle$ in $\mathbf{S}$. Consider now a *structural invariant*, say $\mathcal{I}$, having the form $\mathbf{p}^T \cdot \mathbf{x} = \mathbf{p}^T \cdot \mathbf{m}_0$.

Our heuristics works by removing an edge e of $\mathbf{S}$, whenever we can prove that the markings in $elem_e(\mathbf{S})$ denote cones that do not intersect the structural invariant $\mathcal{I}$. To check this condition on the edge e connecting a node of layer i and a node of layer $i+1$, we first compute the two values $min_{\prec}(e)$ and $min_{\succ}(e)$ defined as follows: $min_{\prec}(e)$ is the minimal value of *prefixes* $\langle m_1, \ldots, m_i\rangle$ of tuples in $elem_e(\mathbf{S})$ evaluated on the function $\mathbf{x} \rightsquigarrow \mathbf{p}^T \cdot \mathbf{x}$; symmetrically, $min_{\succ}(e)$ is computed for *suffixes* $\langle m_{i+1}, \ldots, m_k\rangle$. Specifically, we define

$$min_{\prec}(e) = min\ \{\ \mathbf{p}^T \cdot \langle m_1, \ldots, m_i, 0, \ldots, 0\rangle \mid \langle m_1, \ldots, m_i, \ldots, m_k\rangle \in elem_e(\mathbf{S})\}$$
$$min_{\succ}(e) = min\ \{\ \mathbf{p}^T \cdot \langle 0, \ldots, 0, m_{i+1}, \ldots, m_k\rangle \mid \langle m_1, \ldots, m_{i+1}, \ldots, m_k\rangle \in elem_e(\mathbf{S})\}.$$

The following two properties characterize our heuristics.

Theorem 31 Given an initial marking $\mathbf{m}_0$, a k-ucCST $\mathbf{S}$, a structural property $\mathbf{p}^T \cdot \mathbf{x} = \mathbf{p}^T \cdot \mathbf{m}_0$ with $\mathbf{p}^T \cdot \mathbf{m}_0 \in \mathbb{N}$, and an edge e of $\mathbf{S}$, if $min_{\prec}(e) + min_{\succ}(e) \ > \ \mathbf{p}^T \cdot \mathbf{m}_0$, then $cone(\mathbf{m}) \cap \mathcal{O}(\mathbf{m}_0, \mathbf{p}) = \emptyset$ for any $\mathbf{m} \in elem_e(\mathbf{S})$.

Theorem 32 Given a k-ucCST $\mathbf{S}$, and a structural invariant $\mathbf{p}^T.\mathbf{x} = \mathbf{p}^T.\mathbf{m_0}$ such that $\mathbf{p}^T.\mathbf{m_0} \in \mathbb{N}$, the computation of $min_{\prec}$ and $min_{\succ}$ for each $e = \langle n, n'\rangle$ ($n, n' \in N^{\mathbf{S}}$) such that $n' \in succ(n)$ can be done in time $\mathcal{O}(|\mathbf{S}|)$.

Proof. Algorithm 22 computes $min_{\succ}$ on a k-ucCST $\mathbf{S}$ for all $e = \langle n, n'\rangle$ such that $n' \in succ^{\mathbf{S}}(n)$.

The algorithm is linear in the size of the input k-ucCST. In fact, it amounts to a breadth-first search. The algorithm that computes $min_{\prec}$ can be defined starting from the algorithm computing $min_{\succ}$ by replacing the relation $succ^{\mathbf{S}}$ by the relation $father^{\mathbf{S}}$, $min_{\succ}$ by $min_{\prec}$ and traversing the k-ucCST from the first layer to the last one. □

Based on these results, Algorithm 23 approximates the removing of the *useless* paths. After removing arcs from the k-ucCST $\mathbf{S}$, $\mathbf{S}$ could contain useless nodes n (i.e. with $father^{\mathbf{S}}(n) = \emptyset$

Algorithm 22: $\mathsf{min}_{\succ}$

Input: $\mathbf{S}$: k-ucCST, $\mathbf{p}$: place invariant
begin
  foreach $e = \langle n, n' \rangle$ *such that* $n' \in succ^{\mathbf{S}}(n)$ **do**
    $min_{\succ}(e) \leftarrow 0$;
  $minprec(end^{\mathbf{S}}) \leftarrow 0$;
  foreach $i : k$ *to* 1 $(bottom{-}up)$ **do**
    foreach *node* $n \in N_i^{\mathbf{S}}$ **do**
      $minprec(n) \leftarrow \infty$;
      foreach $e = \langle n, n' \rangle$ *s.t.* $n' \in succ^{\mathbf{S}}(n)$ **do**
        $min_{\succ}(e) \leftarrow minprec(n')$;
        $minprec(n) \leftarrow min(minprec(n), minprec(n'))$;
      $minprec(n) \leftarrow minprec(n) + \mathbf{p}_i \cdot val(n)$;
  foreach $e = \langle root^{\mathbf{S}}, n \rangle$ *such that* $n \in succ^{\mathbf{S}}(root^{\mathbf{S}})$ **do**
    $min_{\succ}(e) \leftarrow minprec(n)$;
end

or $succ^{\mathbf{S}}(n) = \emptyset$) and condition 6 of Definition 42 could not be satisfied. The useless nodes are removed by applying a bottom-up traversal of $\mathbf{S}$ that removes the nodes n such that $succ^{\mathbf{S}}(n) = \emptyset$ and a top-to-bottom traversal that removes the nodes n such that $father^{\mathbf{S}}(n) = \emptyset$. Condition 6 of Definition 42 is ensured by applying Algorithm 28 shown in appendix. Since the traversals and Algorithm 28 are applied in polynomial time, the removing of useless paths by using Algorithm 23 is done in polynomial time.

Algorithm 23: ApproxSH

Input: $\mathbf{S}$: k-ucCST, $\mathbf{p}$: place invariant, $\mathbf{m}_0$: marking
begin
  $\mathsf{min}_{\succ}(\mathbf{S}, \mathbf{p})$;
  $\mathsf{min}_{\prec}(\mathbf{S}, \mathbf{p})$;
  Remove each $e = \langle n, n' \rangle$ *such that* $\min_{\succ}(e) + min_{\prec}(e) > \mathbf{p}^T \cdot \mathbf{m_0}$;
  Remove nodes n *such that* $succ^{\mathbf{S}}(n) = 0$ *or* $father^{\mathbf{S}}(n) = \emptyset$;
  Adjust_Condition6($\mathbf{S}$) ;
end

As an example of application of the structural heuristics, consider the k-ucCST of Figure 5.14. The k-ucCST $\mathbf{S}$ contains the elements obtained at iteration 1, the pairs of values on the arcs are the values $min_{\prec}(e)$ and $min_{\succ}(e)$ for the place invariant $x_2 + x_5 = 1$, the dashed edges can be removed and thus the useless element $\langle 1, 1, 1, 0, 1 \rangle$ is removed from the k-ucCST. Note that if we use the invariant $x_3 + x_4 = 1$ then the last element can also be eliminated. The heuristics simply traverses a k-ucCST layer by layer, removing all edges that satisfy the hypothesis of Theorem 31.

If we have several place invariants, Algorithm 23 is applied successively for each of them.

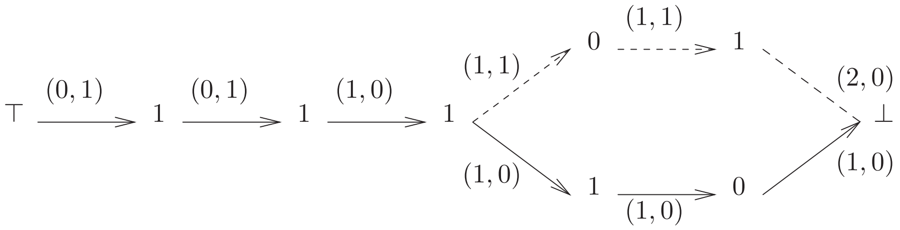

Figure 5.14: Application of Heuristics for Structural Invariants to a k-ucCST.

Algorithm 24: $\mathsf{Symbolic}_{\mathsf{BS}}$

Input: $\mathbf{S}_U$: k-ucCST, $\mathbf{m}_0$: *marking*
Result: *boolean*
begin
  $\mathbf{S} \leftarrow \mathbf{S}_U$;
  $\mathbf{T} \leftarrow$ k-ucCST_Empty_set;
  while *not*(k-ucCST_Subsumption($\mathbf{S}, \mathbf{T}$)) **do**
    $\mathbf{T} \leftarrow$ k-ucCST_Union($\mathbf{T}, \mathbf{S}$);
    $\mathbf{R} \leftarrow$ k-ucCST_Empty_set;
    foreach *transition* t **do**
      $\mathbf{N} \leftarrow$ k-ucCST_Pre($\mathbf{S}, t$);
      k-ucCST_Structural_Reduction($\mathbf{N}$);
      k-ucCST_Heuristics_Minimisation($\mathbf{N}, \mathbf{R}, \mathbf{T}$);
      k-ucCST_Heuristic_Normal_Form($\mathbf{N}$);
      $\mathbf{R} \leftarrow$ k-ucCST_Union($\mathbf{N}, \mathbf{R}$);
    $\mathbf{S} \leftarrow \mathbf{R}$;
  return k-ucCST_Sat($\mathbf{m}_0, \mathbf{T}$);
end

Notice that, in that case, after the application of the algorithm for each place invariant, if we apply again the algorithm for the considered place invariants, it could still remove edges. Let us consider the Petri net of Figure 5.15(a) with the initial marking $\langle 0,0,0,0,1,1\rangle$ where the *ith* component ($1 \leq i \leq 5$) corresponds to the place p_i. That Petri net has two (minimal) place invariants $\mathbf{p}_1 = \langle 0,1,0,0,1,0\rangle$ and $\mathbf{p}_2 = \langle 0,0,0,1,0,1\rangle$. Let us consider the k-ucCST of Figure 5.15(b). If we apply Algorithm 23 on that k-ucCST for $\mathbf{p}_1$, we obtain the k-ucCST represented by Figure 5.15(c). If we then apply Algorithm 23 for $\mathbf{p}_2$, we obtain the k-ucCST of Figure 5.15(d). From this k-ucCST, if we apply again the algorithm for $\mathbf{p}_1$, it will remove paths again and construct the k-ucCST of Figure 5.15(e).

Based on the previous ideas, the k-ucCST-based symbolic reachability algorithms (Algorithm 11 and Algorithm 12) can be enriched to obtain Algorithm 24 and Algorithm 25. Given a set of place invariants computed a priori via a static analysis of a Petri Net, the procedure k-ucCST_Structural_Reduction implements the *reduction algorithm* based on structural heuristics of Theorem 30 or the approximation presented in this section to removes *useless* generators from the input k-ucCST. To complete the scenario, we need to compute *automatically* the structural invariants. This can be done using specialized libraries to compute

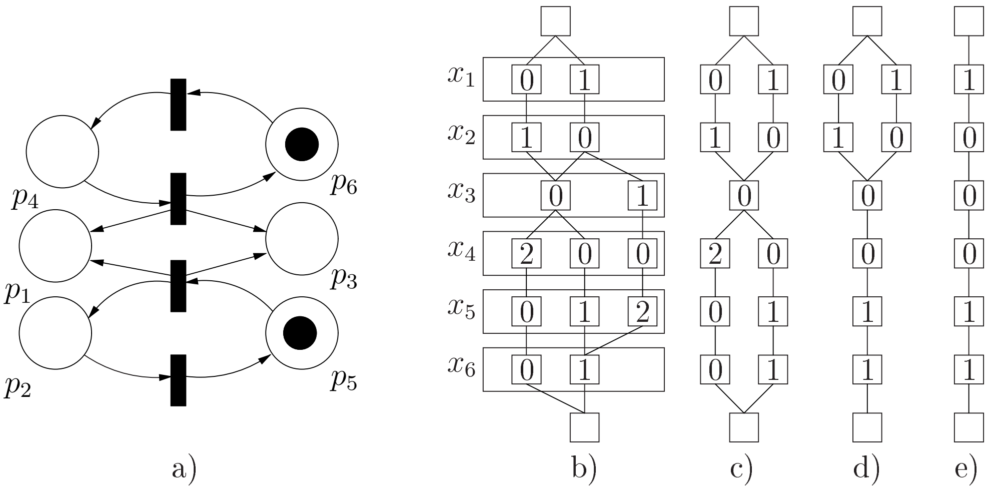

Figure 5.15: An example where it is interesting to apply several times Algorithm 23 for the same place-invariants.

Algorithm 25: $\mathsf{Symbolic}_{\mathsf{BS}}$

Input: $\langle \mathcal{P}, \mathcal{B} \rangle$: Petri net, $\mathbf{S}_U$: k-ucCST, $\mathbf{m}_0$: marking
Result: *boolean*
begin
    $\mathbf{S} \leftarrow \mathbf{S}_U$;
    $\mathbf{T} \leftarrow$ k-ucCST_Empty_set;
    while *not*(k-ucCST_Empty($\mathbf{S}$)) **do**
        $\mathbf{T} \leftarrow$ k-ucCST_Union($\mathbf{T}, \mathbf{S}$);
        $\mathbf{R} :=$ k-ucCST_Empty_set;
        foreach $M \in \mathcal{B}$ **do**
            $\mathbf{N} \leftarrow$ k-ucCST_Pre($\mathbf{S}, M$);
            k-ucCST_Structural_Reduction($\mathbf{N}$);
            k-ucCST_Minimization($\mathbf{N}, \mathbf{R}$);
            k-ucCST_Minimization($\mathbf{R}, \mathbf{N}$);
            k-ucCST_Minimization($\mathbf{N}, \mathbf{T}$);
            k-ucCST_Minimization($\mathbf{T}, \mathbf{N}$);
            k-ucCST_NormalForm($\mathbf{N}$);
            $\mathbf{R} \leftarrow$ k-ucCST_Union($\mathbf{N}, \mathbf{R}$);
        $\mathbf{S} \leftarrow \mathbf{R}$;
    return k-ucCST_Sat($\mathbf{m}_0, \mathbf{T}$);
end

---

systems of generators for place invariants like the one available with GreatSPN [CFGR95], or PEP [Gra97a].

Extension of the Structural Heuristics to MTNs

We present in this section how to extend the structural heuristics to the general class of MTN. More precisely, we show how to compute from a MTN $\mathcal{M}$ with n places a set of tuples $\mathbf{p} \in \mathbb{N}^n$ such that $\mathsf{Reach}_{\mathcal{M}}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m}, \mathbf{p})$ for any marking $\mathbf{m}$. Having a set of such tuples, the structural heuristics presented in the previous sections can then be used to prune the search space when applying the backward approach. Intuitively, we encode MTN into a general Petri net extension called *Self-Modifying Petri Nets*. Then, we use a result given by G. Ciardo in [Cia94] (Theorem 33) to compute tuples $\mathbf{p} \in \mathbb{N}^n$ such that $\mathsf{Reach}_{\mathcal{M}}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m}, \mathbf{p})$ for any marking $\mathbf{m}$.

Self-modifying Petri nets are defined as follows.

Definition 68 (Self-Modifying Petri nets [Val78]) A *self-modifying Petri net* is a tuple $\langle \mathcal{P}, T, D^-, D^+ \rangle$ where

- $\mathcal{P}$ is a finite set of places;
- T is a finite set of transitions;
- $\forall 1 \leq i \leq |T|, \forall 1 \leq j \leq |\mathcal{P}| : D^-_{ij} : \mathbb{N}^{|\mathcal{P}|} \rightarrow \mathbb{N}$ is a linear expression of the form $\alpha^-_{ij} + \sum_{k=1..|\mathcal{P}|} \beta^-_{ijk} \cdot x_k$ where $\alpha^-_{ij} \in \mathbb{N}$ and $\beta^-_{ijk} \in \mathbb{N}$ for all $1 \leq i \leq | T |$ and $1 \leq j, k \leq | \mathcal{P} |$;
- $\forall 1 \leq i \leq |T|, \forall 1 \leq j \leq |\mathcal{P}| : D^+_{ij} : \mathbb{N}^{|\mathcal{P}|} \rightarrow \mathbb{N}$ is a linear expression of the form $\alpha^+_{ij} + \sum_{k=1..|\mathcal{P}|} \beta^+_{ijk} \cdot x_k$ where $\alpha^+_{ij} \in \mathbb{N}$ and $\beta^+_{ijk} \in \mathbb{N}$ for all $1 \leq i \leq | T |$ and $1 \leq j, k \leq | \mathcal{P} |$.

As expected, the variable x_k in the linear expressions labeling the arcs of self-modifying nets describes the number of tokens contained by the place p_k.

A transition t_j is enabled in a marking $\mathbf{m}$ if, for all $p_i \in \mathcal{P}$, we have $\mathbf{m}(p_i) \geq D^-_{ij}(\mathbf{m})$. Firing an enabled transition t_j from $\mathbf{m}$ leads to the marking $\mathbf{m}'$ (noted $\mathbf{m} \rightarrowtail^{t_j} \mathbf{m}'$) such that for all $p_i \in \mathcal{P}$, we have $\mathbf{m}'(p_i) = \mathbf{m}(p_i) + D^+_{ij}(\mathbf{m}) - D^-_{ij}(\mathbf{m})$.

Figure 5.16(a) shows an example of a self-modifying Petri nets. Place and transitions are represented in the same fashion than for MTN. An arc from a transition t_i to a place p_j is labeled by the linear expression D^-_{ij} and an arc from a place p_j to a transition t_i is labeled by the linear expression D^+_{ij}. If there is no arc between a place and a transition, this means that the linear expression is the constant expression 0. For instance, the arcs from p_4 to t_2 and from t_2 to p_3 in the net of Figure 5.16(a) models a (unbounded) transfer of tokens. When t_2 is fired, the k tokens in p_4 are removed from p_4 and k tokens are added into p_3.

The token flow matrix of a self-modifying net is a matrix where the elements $c_{ij} = D^+_{ij} - D^-_{ij}$. The notion of place invariants is extended to self-modifying Petri nets in the obvious way. For instance, $\langle 1, 1, 0, 0 \rangle$ is the only (minimal) place invariant of the self-modifying Petri net shown in Figure 5.16(a). The following theorem, given by C. Ciardo in [Cia94], gives an algorithm to compute place invariants of self-modifying nets.

Theorem 33 (From [Cia94]) Let $\mathcal{M} = \langle \mathcal{P}, T, D^-, D^+ \rangle$ be a self-modifying Petri net and $N = \langle \mathcal{P}, \mathcal{B} \rangle$ be a Petri net where $\mathcal{B} = \{M_{t_j} = \langle \langle \mathcal{I}, \mathcal{O} \rangle, \emptyset \rangle | t_j \in T, \forall p_i \in \mathcal{P} : \mathcal{I}(p_i) = \alpha^-_{ij}, \mathcal{O}(p_i) = \alpha^+_{ij}\} \cup \{M_{t_{(i,j)}} = \langle \langle \mathcal{I}, \mathcal{O} \rangle, \emptyset \rangle | 1 \leq i \leq |T|, 1 \leq j \leq |\mathcal{P}|, \forall p_k \in \mathcal{P} : \mathcal{I}(p_k) = \beta^-_{ijk}, \mathcal{O}(p_k) = \beta^+_{ijk}, \}$.

Then, $\mathcal{M}$ and N have the same set of place invariants.

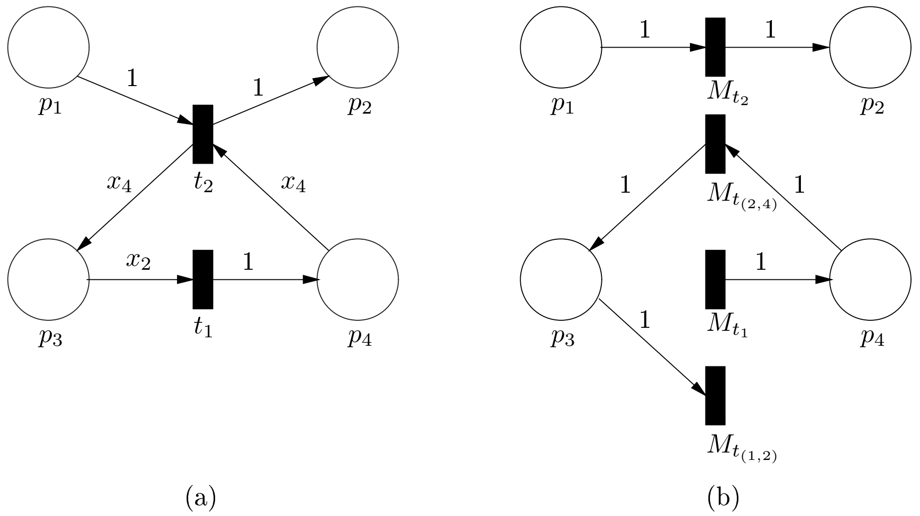

Figure 5.16: (a) an example of a self-modifying Petri net, (b) the Petri net constructed following Theorem 33 from the self-modifying Petri net shown in (a).

Figure 5.16(b) shows the Petri net constructed following Theorem 33 from the self-modifying Petri net of Figure 5.16(a).

We now focus on the problem of computing tuples $\mathbf{p} \in \mathbb{N}^n$ for a MTN $\mathcal{M}$ with n places such that $\mathsf{Reach}_{\mathcal{M}}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m}, \mathbf{p})$ for any marking $\mathbf{m}$. This is achieved by constructing from $\mathcal{M}$ a self-modifying Petri net N such that the place invariants of N are such tuples.

Mainly, there are two problems to encode MTNs into self-modifying Petri nets:

- There is no notion of bounds (of transfers) in self-modifying Petri nets, i.e. we can encode unbounded transfers (see example of Figure 5.16) but not bounded transfers;
- the effect of firing a transition in self-modifying Petri nets are computed in one step and the effect of firing a transition in MTNs is computed in tree steps (see Definition 11).

To avoid the first difficulty, we proceed as follows.

Let $\mathcal{M}$ be a MTN with n places. We define the MTN $\mathcal{M}^{+\infty}$ as the MTN constructed from $\mathcal{M}$ by replacing all the bounds of the transfer blocks of $\mathcal{M}$ by $+\infty$. That corresponds to remove the bounds from the multi-transfers of $\mathcal{M}$. The following proposition says that the bounds of multi-transfers of $\mathcal{M}$ can be removed to compute tuples $\mathbf{p} \in \mathbb{N}^n$ such that $\mathsf{Reach}_{\mathcal{M}}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m}, \mathbf{p})$ for any marking $\mathbf{m}$, i.e. the invariants do not depend on the bounds.

In the following, for a tuple $\mathbf{p} \in \mathbb{N}^n$, we sometimes denote the value of the dimension corresponding to a place p in $\mathbf{p}$ by $\mathbf{p}(p)$.

Proposition 23 Let $\mathcal{M}$ be a MTN with n places and let $\mathbf{p} \in \mathbb{N}^n$.

Then, the following holds.

$$\forall \mathbf{m} \in \mathbb{N}^n : \mathsf{Reach}_{\mathcal{M}}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m}, \mathbf{p}) \text{ iff } \forall \mathbf{m} \in \mathbb{N}^n : \mathsf{Reach}_{\mathcal{M}^{+\infty}}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m}, \mathbf{p})$$

Proof. In the following, we suppose that $\mathcal{M} = \langle \mathcal{P}, \mathcal{B} \rangle$ and $\mathcal{M}^{+\infty} = \langle \mathcal{P}, \mathcal{B}^{+\infty} \rangle$ such that for any $M \in \mathcal{B}$, we have $M^{+\infty} \in \mathcal{B}^{+\infty}$ constructed from M by replacing the bounds of the transfer blocks by $+\infty$.

$\Rightarrow$ Per absurdum, suppose that $\mathbf{p}$ is such that $\mathsf{Reach}_{\mathcal{M}}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m}, \mathbf{p})$ for all $\mathbf{m} \in \mathbb{N}^n$ and there exist $\mathbf{m}, \mathbf{m}' \in \mathbb{N}^n$ and $M^{+\infty} \in \mathcal{B}^{+\infty}$ with

$$\mathbf{m} \rightarrowtail^{M^{+\infty}} \mathbf{m}' \text{ and } \mathbf{p}^T \cdot \mathbf{m} \neq \mathbf{p}^T \cdot \mathbf{m}' \tag{5.3}$$

i.e. $\mathsf{Reach}_{\mathcal{M}^{+\infty}}(\mathbf{m}) \not\subseteq \mathcal{O}(\mathbf{m}, \mathbf{p})$.

Assume that $M^{+\infty} = \langle \langle \mathcal{I}, \mathcal{O} \rangle, \{S\} \rangle$ and $S = \langle \langle P_1, p_1 \rangle, \ldots, \langle P_k, p_k \rangle, +\infty \rangle$. Let us construct $\mathbf{m}_1$ from $\mathbf{m}$ such that $\mathbf{m}_1(p) = \mathcal{I}(p)$ if $p \in \cup_{1 \leq i \leq k} P_i$, $\mathbf{m}(p)$ otherwise. $M^{+\infty}$ can still be fired from $\mathbf{m}_1$ and we have $\mathbf{m}_1 \rightarrowtail^{M^{+\infty}} \mathbf{m}'_1$. As no token is transfered, it follows by hypothesis that

$$\mathbf{m}_1 \rightarrowtail^{M} \mathbf{m}'_1 \text{ and } \mathbf{p}^T \cdot \mathbf{m}_1 = \mathbf{p}^T \cdot \mathbf{m}'_1 \tag{5.4}$$

We also have

$$\mathbf{p}^T \cdot \mathbf{m} = \mathbf{p}^T \cdot \mathbf{m}_1 + \mathbf{p}^T \cdot \overline{\mathbf{m}_1} \text{ and } \mathbf{p}^T \cdot \mathbf{m}' = \mathbf{p}^T \cdot \mathbf{m}'_1 + \mathbf{p}^T \cdot \overline{\mathbf{m}'_1} \tag{5.5}$$

where $\overline{\mathbf{m}_1} = \mathbf{m} - \mathbf{m}_1$ and $\overline{\mathbf{m}'_1} = \mathbf{m}' - \mathbf{m}'_1$. From (5.3)-(5.4)-(5.5), we obtain that $\mathbf{p}^T \cdot \overline{\mathbf{m}_1} \neq \mathbf{p}^T \cdot \overline{\mathbf{m}'_1}$. As for all $1 \leq i \leq k : \overline{\mathbf{m}}_1(P_i) = \overline{\mathbf{m}}'_1(p_i)$ (remember that we have $(\{p_i\} \cup P_i) \cap (\{p_j\} \cap P_j) = \emptyset$ for all $i \neq j$, hence all the tokens in P_i are transferred in p_i when firing M) and for all $p \notin \bigcup_{i=1\ldots k}(P_i \cup \{p_i\}) : \overline{\mathbf{m}}_1(p) = \overline{\mathbf{m}}'_1(p)$, there must exists i and $p \in P_i$ with $\mathbf{p}(p) \neq \mathbf{p}(p_i)$.

Suppose that we have such places p and p_i. Let us define $\mathbf{m}_p$ and $\mathbf{m}_{p_i}$ such that $\mathbf{m}_p(p') = 1$ if $p' = p$, 0 otherwise, $\mathbf{m}_{p_i}(p') = 1$ if $p' = p_i$, 0 otherwise and $\overline{\mathbf{m}}_2$ and $\overline{\mathbf{m}}'_2$ such that $\overline{\mathbf{m}}_2 = \mathbf{m}_1 + \mathbf{m}_p$ and $\overline{\mathbf{m}}'_2 = \mathbf{m}'_1 + \mathbf{m}_{p_i}$. We have $\overline{\mathbf{m}}_2 \rightarrowtail^{M} \overline{\mathbf{m}}'_2$ in $\mathcal{M}$ with $\mathbf{p}^T \cdot \mathbf{m}_1 + \mathbf{p}^T \cdot \mathbf{m}_p = \mathbf{p}^T \cdot \mathbf{m}'_1 + \mathbf{p}^T \cdot \mathbf{m}_{p_i}$. Finally, we obtain $\mathbf{p}^T \cdot \mathbf{m}_p = \mathbf{p}^T \cdot \mathbf{m}_{p_i}$, hence $\mathbf{p}(p) = \mathbf{p}(p_i)$, and that makes a contradiction.

$\Leftarrow$ Per absurdum, suppose that $\mathbf{p}$ is such that $\mathsf{Reach}_{\mathcal{M}^{+\infty}}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m}, \mathbf{p})$ for all $\mathbf{m} \in \mathbb{N}^n$ and there exists a markings $\mathbf{m}$ and a multi-transfer $M \in \mathcal{B}$ with

$$\mathbf{m} \rightarrowtail^{M} \mathbf{m}' \text{ and } \mathbf{p}^T \cdot \mathbf{m} \neq \mathbf{p}^T \cdot \mathbf{m}' \tag{5.6}$$

Let $\mathbf{m}^{\mathcal{I}} = \mathbf{m} - \mathcal{I}$ and rem_l for $1 \leq l \leq k$ is equal to the number of tokens removed from p by the transfer $\langle P_l, p_l \rangle$ when applying the multi-transfer M from $\mathbf{m}$ to reach $\mathbf{m}'$. We define $\overline{\mathbf{m}}$ such that $\overline{\mathbf{m}}(p) = \mathbf{m}^{\mathcal{I}}(p) - \sum_{l=1\ldots k} rem_l(p)$ for all $p \in \mathcal{P}$. Intuitively, $\overline{\mathbf{m}}$ describes the tokens that are not removed by Petri net arcs and not transfered by transfers when applying M on $\mathbf{m}$. Let $\mathbf{m}_1 = \mathbf{m} - \overline{\mathbf{m}}$. By construction, we have $\mathbf{m}_1 \rightarrowtail^{M} \mathbf{m}'_1$ where $\mathbf{m}'_1 = \mathbf{m}' - \overline{\mathbf{m}}$ and $\mathbf{m}_1 \rightarrowtail^{M^{+\infty}} \mathbf{m}'_1$. Finally, we have from (5.6) that $\mathbf{p}^T \cdot \mathbf{m}_1 \neq \mathbf{p}^T \cdot \mathbf{m}'_1$. We conclude that $\mathsf{Reach}_{\mathcal{M}^{+\infty}}(\mathbf{m}_1) \not\subseteq \{\mathbf{m} | \mathbf{p}^T \cdot \mathbf{m}_1 = \mathbf{p}^T \cdot \mathbf{m}\}$. This makes a contradiction with the fact that $\mathsf{Reach}_{\mathcal{M}^{+\infty}}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m}, \mathbf{p})$. □

The second problem to encode MTNs into self-modifying Petri nets is avoided by separating the Petri net part from the transfer part of each multi-transfer. More precisely, given a MTN $\mathcal{M} = \langle \mathcal{P}, \mathcal{B} \rangle$ we define the MTN $\mathcal{M}^S = \langle \mathcal{P}, \mathcal{B}^S \rangle$ such that $\mathcal{B}^S = \bigcup_{M \in \mathcal{B}} \{\{M^{\frown}, M^t\} | M = \langle \langle \mathcal{I}, \mathcal{O} \rangle, \{S_1, \ldots, S_n\} \rangle$ with $M^{\frown} = \langle \langle \mathcal{I}_0, \mathcal{O}_0 \rangle, \{S_1, \ldots, S_n\} \rangle$ and $M^t = \langle \langle \mathcal{I}, \mathcal{O} \rangle, \emptyset \rangle \}$. The main interest of $\mathcal{M}^S$ is stated by the following proposition.

Proposition 24 Let $\mathcal{M}$ be a MTN with n places and let $\mathbf{p} \in \mathbb{N}^n$.

Then, the following holds.

$$\forall \mathbf{m} \in \mathbb{N}^n : \mathsf{Reach}_{\mathcal{M}}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m}, \mathbf{p}) \text{ iff } \forall \mathbf{m} \in \mathbb{N}^n : \mathsf{Reach}_{\mathcal{M}^S}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m}, \mathbf{p})$$

Proof. In the following, we suppose that $M = \langle\langle\mathcal{I},\mathcal{O}\rangle,\{S_1,\ldots,\mathcal{S}_l\}\rangle$, $M^t = \langle\langle\mathcal{I},\mathcal{O}\rangle,\emptyset\rangle$, $M^\frown = \langle\langle\mathcal{I}_0,\mathcal{O}_0\rangle,\{S_1,\ldots,S_l\}\rangle$ with $S_i = \langle\langle P_{i1},p_{i1}\rangle,\ldots,\langle P_{ik_i},p_{ik_i}\rangle,c_i\rangle, \forall 1 \le i \le l$.
$\Rightarrow$ Per absurdum, suppose that $\mathbf{p}$ is such that $\mathsf{Reach}_{\mathcal{M}}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m},\mathbf{p})$ for any $\mathbf{m}$ but we do not have $\mathsf{Reach}_{\mathcal{M}^S}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m},\mathbf{p})$ for any marking $\mathbf{m}$. We consider two cases.

(i) In the first case, we have

$$\exists M^t \in \mathcal{B}^S : \exists \mathbf{m},\mathbf{m}' \in \mathbb{N}^n : \mathbf{m} \rightarrowtail^{M^t} \mathbf{m}' \text{ with } \mathbf{p}^T\cdot\mathbf{m} \neq \mathbf{p}^T\cdot\mathbf{m}' \tag{5.7}$$

Let $\mathbf{m}_1$ be a marking such that $\mathbf{m}_1(p) = \mathcal{I}(p)$ if $p \in \cup_{1\le i\le l}\cup_{1\le j\le k_i} P_{ij}$, $\mathbf{m}_1(p) = \mathbf{m}(p)$ otherwise. Let $\overline{\mathbf{m}} = \mathbf{m} - \mathbf{m}_1$. We have $\mathbf{m}_1 \rightarrowtail^M \mathbf{m}_1'$ in $\mathcal{M}$ with $\mathbf{m}_1' = \mathbf{m}' - \overline{\mathbf{m}}$ (there is not actual transfer). It follows that

$$\mathbf{p}^T\cdot\mathbf{m} = \mathbf{p}^T\cdot\mathbf{m}_1 + \mathbf{p}^T\cdot\overline{\mathbf{m}} \text{ and } \mathbf{p}^T\cdot\mathbf{m}' = \mathbf{p}^T\cdot\mathbf{m}_1' + \mathbf{p}^T\cdot\overline{\mathbf{m}} \tag{5.8}$$

From (5.7) and (5.8) we obtain $\mathbf{p}^T\cdot\mathbf{m}_1 \neq \mathbf{p}^T\cdot\mathbf{m}_1'$. That implies that $\mathsf{Reach}_{\mathcal{M}}(\mathbf{m}_1) \not\subseteq \{\mathbf{m}|\mathbf{p}^T\cdot\mathbf{m}_1 = \mathbf{p}^T\cdot\mathbf{m}\}$. That makes a contradiction with the fact that $\mathsf{Reach}_{\mathcal{M}}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m},\mathbf{p})$ for any $\mathbf{m}$.

(ii) In the other case, we have

$$\exists M^\frown \in \mathcal{B}^S : \exists \mathbf{m},\mathbf{m}' : \mathbf{m} \rightarrowtail^{M^\frown} \mathbf{m}' \text{ with } \mathbf{p}^T\cdot\mathbf{m} \neq \mathbf{p}^T\cdot\mathbf{m}' \tag{5.9}$$

Let us construct $\mathbf{m}_1$ such that $\mathbf{m}_1 = \mathbf{m}+\mathcal{I}$ and suppose that we have $\mathbf{m}_1 \rightarrowtail^M \mathbf{m}_1'$ in $\mathcal{M}$ where $\mathbf{m}_1' = \mathbf{m}'+\mathcal{O}$. From (5.9) we obtain $\mathbf{p}^T\cdot\mathbf{m}_1 - \mathbf{p}^T\cdot\mathcal{I} \neq \mathbf{p}^T\cdot\mathbf{m}_1' - \mathbf{p}^T\cdot\mathcal{O}$. Assuming that for any transition M_t there is no $\mathbf{m},\mathbf{m}'$ with $\mathbf{m} \rightarrowtail^{M_t} \mathbf{m}'$ and $\mathbf{p}^T\cdot\mathbf{m} \neq \mathbf{p}^T\cdot\mathbf{m}'$, we have $\mathbf{p}^T\cdot\mathcal{I} = \mathbf{p}^T\cdot\mathcal{O}$ (as $\mathcal{I} \rightarrowtail^{M^t} \mathcal{O}$). It follows that $\mathbf{p}^T\cdot\mathbf{m}_1 \neq \mathbf{p}^T\cdot\mathbf{m}_1'$. This implies that $\mathsf{Reach}_{\mathcal{M}}(\mathbf{m}_1) \not\subseteq \{\mathbf{m}|\mathbf{p}^T\cdot\mathbf{m}_1 = \mathbf{p}^T\cdot\mathbf{m}\}$. That makes a contradiction with the fact that $\mathsf{Reach}_{\mathcal{M}}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m},\mathbf{p})$ for any $\mathbf{m}$.

$\Leftarrow$ Per absurdum, suppose that $\mathbf{p}$ is such that $\mathsf{Reach}_{\mathcal{M}^S}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m},\mathbf{p})$ for any $\mathbf{m}$ but there exist $M \in \mathcal{B}$ and $\mathbf{m},\mathbf{m}' \in \mathbb{N}^n$ with

$$\mathbf{m} \rightarrowtail^M \mathbf{m}' \text{ and } \mathbf{p}^T\cdot\mathbf{m} \neq \mathbf{p}^T\cdot\mathbf{m}' \tag{5.10}$$

Let us construct the marking $\mathbf{m}_1$ such that $\mathbf{m}_1(p) = \mathcal{I}(p)$ if $p \in \cup_{1\le i\le l}\cup_{1\le j\le k_i} P_{ij}$, $\mathbf{m}_1(p) = \mathbf{m}(p)$ otherwise and let $\mathbf{m}_1'$ such that

$$\mathbf{m}_1 \rightarrowtail^{M^t} \mathbf{m}_1' \text{ with } \mathbf{p}^T\cdot\mathbf{m}_1 = \mathbf{p}^T\cdot\mathbf{m}_1' \tag{5.11}$$

Let $\overline{\mathbf{m}} = \mathbf{m} - \mathbf{m}_1$ and $\overline{\mathbf{m}}' = \mathbf{m}' - \mathbf{m}_1'$. It follows from (5.10) that $\mathbf{p}^T\cdot\overline{\mathbf{m}} + \mathbf{p}^T\cdot\mathbf{m}_1 \neq \mathbf{p}^T\cdot\overline{\mathbf{m}}' + \mathbf{p}^T\cdot\mathbf{m}_1'$. As we have $\overline{\mathbf{m}} \rightarrowtail^{M^\frown} \overline{\mathbf{m}'}$ we obtain by hypothesis that $\mathbf{p}^T\cdot\overline{\mathbf{m}} = \mathbf{p}^T\cdot\overline{\mathbf{m}}'$ and $\mathbf{p}^T\cdot\mathbf{m}_1 \neq \mathbf{p}^T\cdot\mathbf{m}_1'$. This makes a contradiction w.r.t. (5.11). □

Given a MTN $\mathcal{M}$ with n places, to compute tuples $\mathbf{p} \in \mathbb{N}^n$ such that $\mathsf{Reach}_{\mathcal{M}}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m},\mathbf{p})$ for any marking $\mathbf{m}$, we proceed as follows. First, we replace the bounds of the multi-transfers of $\mathcal{M}$ by $+\infty$, then we separate the Petri net part from the transfer part of each multi-transfer. Following Proposition 23 and Proposition 24, this leads to a MTN $\mathcal{M}'$ with n places such that for any $\mathbf{p} \in \mathbb{N}^n$, we have $\mathsf{Reach}_{\mathcal{M}}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m},\mathbf{p})$ if and only if $\mathsf{Reach}_{\mathcal{M}'}(\mathbf{m}) \subseteq \mathcal{O}(\mathbf{m},\mathbf{p})$. Moreover, $\mathcal{M}'$ can easily be encoded into a self-modifying Petri net N as follows.

First, N has the same set of places than $\mathcal{M}'$. The multi-transfers of the form $\langle\langle\mathcal{I},\mathcal{O}\rangle,\emptyset\rangle$ are encoded through a transition t where the arcs from each place p to t is labelled by the constant function $\mathcal{I}(p)$ and the arcs from t to each place p is labeled by $\mathcal{O}(p)$.

The multi-transfers of the form $\langle\langle\mathcal{I}_0,\mathcal{O}_0\rangle,\{\langle\{B_1,\ldots,B_n\},+\infty\rangle\}\rangle$ with $B_i=\langle P_i,p_i\rangle$ $(1\leq i\leq n)$ are encoded through a transition t as follows

For each place p such that there is $1\leq i\leq n$ with $p\in P_i$ the arc from p to t is labelled by x_p (the variable corresponding to p). For the other places p, the arc from p to t is labelled by the constant function 0. For each place p such that there is $1\leq i\leq n$ with $p=p_i$ the arc from t to p is labelled by the expression $e=\sum_{p_j\in P_i}p_j$. For the other places p the arc from t to p is labelled by the constant function 0.

N is equivalent to $\mathcal{M}'$ in the sense that the two nets have the same set of places and the same transitions (expressed in two different ways. One can easily check that the implied semantics is the same). Hence, the place invariants of N are tuples $\mathbf{p}$ that can be used by the structural heuristics defined in the previous sections when analysing $\mathcal{M}$ with a backward search, i.e. the place invariants $\mathbf{p}$ of N are such that $\mathsf{Reach}_{\mathcal{M}}(\mathbf{m})\subseteq\mathcal{O}(\mathbf{m},\mathbf{p})$ for any marking $\mathbf{m}$.

5.4.2 Combining Forward and Backward Approach

As already mentioned, the forward approach cannot be applied on the full class of MTN to decide upward closed properties. However, if we apply Algorithm 17 on a MTN $\mathcal{M}$ with the initial marking $\mathbf{m}_0$, it will compute an over-approximation of the reachable markings that could be used to prune the backward search. More precisely, at the end of the computation of the algorithm, the set $fixpoint$ contains ω-markings such that $\mathsf{Reach}_{\mathcal{M}}(\mathbf{m}_0)\subseteq dcones(fixpoint)$. This property is proved by using the same proof than for Lemma 19. Notice that the termination of Algorithm 17 when applied on a MTN instead of a basic Petri net is ensured following the same arguments than those presented in the proof of Lemma18.

Let us now show that the forward approximation computed by Algorithm 17 can give us more information than the structural invariants. The parameterized MTN of the MESI protocol (see [Han93]) presented in Figure 5.17 has only one structural invariant which says that the number of tokens in all the places is constant in all the forward reachable markings. As the place invalid can contain any number of tokens (the protocol starts with $i\geq 1$ tokens in the invalid place), this invariant does not give any information to prune the backward search. But by applying Algorithm 17, we obtain that the reachable markings are covered by the set of ω-markings $\{\langle+\infty,+\infty,+\infty,0\rangle,\langle+\infty,0,+\infty,+\infty\rangle\}$ (where the places are encoded in the markings as $\langle\mathsf{invalid},\mathsf{shared},\mathsf{modified},\mathsf{exclusive}\rangle$). This over-approximation allows us to verify the mutual exclusion property between shared and exclusive. However, other interesting properties cannot be verified with this over-approximation. As an example, consider the mutual exclusion property saying that *at most one token can be in* exclusive. Backward approach can be applied efficiently to answer this question by eliminating during the search all the upward closed sets that do not intersect with the over-approximation computed forwardly. So, the forward over-approximation can be used to prune the backward search.

Given a MTN $\mathcal{M}$ with the (possibly parametric) initial marking $\mathbf{m}_0$, $\mathcal{O}_{\mathsf{CS}}(\mathcal{M},\mathbf{m}_0)$ denotes the set of ω-markings contained into the set $fixpoint$ at the end of Algorithm 17 when applied on the MTN $\mathcal{M}$ and the initial ω-marking corresponding to $\mathbf{m}_0$, i.e. where the parameters have been replaced by $+\infty$ as explained at the beginning of Section 5.2. The relation between structural invariants and $\mathcal{O}_{\mathsf{CS}}(\mathcal{M},\mathbf{m}_0)$ is given by the following proposition.

Proposition 25 Let $\mathcal{M}$ be a MTN with n places, $\mathbf{m}_0$ be an (possibly parametric) initial marking and $\mathbf{p} \in \mathbb{N}^n$ be such that $\mathsf{Reach}_{\mathcal{M}}(\mathbf{m}) \subseteq dcones(\mathcal{O}(\mathbf{m}, \mathbf{p}))$ for any marking $\mathbf{m}$ and $\mathbf{p}^T \cdot \mathbf{m}_0$ is a natural number (i.e. does not contain parameters).

Then, the following holds.

$$dcones(\mathcal{O}_{\mathsf{CS}}(\mathcal{M}, \mathbf{m}_0)) \subseteq dcones(\mathcal{O}(\mathbf{m}_0, \mathbf{p}))$$

Proof. In the rest of the proof, we consider that the parameters in $\mathbf{m}_0$ are replaced by $+\infty$. We prove by induction on the number i of iterations of the main *while* loop of Algorithm 17 that all the ω-markings $\mathbf{m}$ added into $fixpoint$ are such that $\mathbf{p}^T \cdot \mathbf{m} = \mathbf{p}^T \cdot \mathbf{m}_0$.

Base case: $i = 0$. Just before entering the loop for the first time, $fixpoint$ only contains $\mathbf{m}_0$ and the property is trivially verified.

Induction step: $i = n + 1$. Suppose that for all the ω-markings $\mathbf{m}$ added into $fixpoint$ up to the iteration i of the main loop, we have $\mathbf{p}^T \cdot \mathbf{m} = \mathbf{p}^T \cdot \mathbf{m}_0$ and the algorithm is not terminated. Let us prove that it is also the case for the ω-markings added at the $(n + 1)th$ iteration of the main loop. Suppose that $\mathbf{m}$ is the ω-marking treated at the $(n+1)th$ iteration of the main *while* loop and assume that $\mathbf{m} \rightarrow^M \mathbf{n}$ for some multi-transfer $M = \langle\langle \mathcal{I}, \mathcal{O}\rangle, T\rangle$ of $\mathcal{M}$. Since we have $\mathbf{p}^T \cdot \mathbf{m} = \mathbf{p}^T \cdot \mathbf{m}_0$ by induction hypothesis, we also have $\mathbf{p}^T \cdot \mathbf{n} = \mathbf{p}^T \cdot \mathbf{m}_0$. Indeed, suppose that it is not the case and assume that $\mathbf{p}^T \cdot \mathbf{m} = c$ (notice that $c \in \mathbb{N}$ by hypothesis). We define $\mathbf{m}'$ such that $m'_j = m_j$ if $m_j \neq +\infty$, otherwise $m'_j = max(c + 1, \mathcal{I}_j)$ (notice that in that case $p_j = 0$).

By construction, we have $\mathbf{p}^T \cdot \mathbf{m}' = c$ (since $c \neq +\infty$ and $m'_i = m_i$ when $m_i \neq +\infty$). Since $\mathbf{m}'$ is a regular marking, for any marking $\mathbf{n}'$ such that $\mathbf{m}' \rightarrowtail^M \mathbf{n}'$ we have $\mathbf{p}^T \cdot \mathbf{n}' = c$, otherwise we would have $\mathsf{Reach}_{\mathcal{M}}(\mathbf{m}') \not\subseteq dcones(\mathcal{O}(\mathbf{m}', \mathbf{p}))$. Notice that at least one such ω-marking $\mathbf{n}'$ exists since M is firable from $\mathbf{m}'$ by construction. Hence there is no unbounded transfer in M that transfers all the tokens from the ith place of $\mathcal{M}$ with $m_i = +\infty$ to the jth place of $\mathcal{M}$ such that $p_j \neq 0$, otherwise we would have $\mathbf{p}^T \cdot \mathbf{n}' > c$ following Definition 11 and since $+\infty$ has been replace by a value greater than c in $\mathbf{m}'$. So, we can construct a marking $\mathbf{m}''$ from $\mathbf{m}$ by replacing $+\infty$ by natural numbers such that there exists $\mathbf{n}''$ with $\mathbf{m}'' \rightarrowtail^M \mathbf{n}''$ and for all i with $1 \leq i \leq n$ we have $n''_i = n_i$ if $p_i > 0$ (this is achieved by replacing $+\infty$ by c in the source places of bounded transfers with the bound c, i.e. by ensuring that the maximum number of tokens will be transfered). Since $\mathbf{p}^T \cdot \mathbf{n}'' = \mathbf{p}^T \cdot \mathbf{m}_0$ and $n_i = n''_i$ if $p_i > 0$ for all i such that $1 \leq i \leq n$, we conclude that $\mathbf{p}^T \cdot \mathbf{n} = \mathbf{p}^T \cdot \mathbf{m}_0$. That makes a contradiction with the fact that $\mathbf{p}^T \cdot \mathbf{n} \neq \mathbf{p}^T \cdot \mathbf{m}_0$. We conclude that $\mathbf{p}^T \cdot \mathbf{n} = \mathbf{p}^T \cdot \mathbf{m}_0$ and if $\mathbf{n}$ is put into the tree or is forgotten because there is already a bigger ω-marking in $fixpoint$, we are done.

Otherwise, suppose that we have $\mathbf{m}' \prec \mathbf{n}$ and $\mathsf{Acceleration}$ is applied on $\mathbf{m}'$ and $\mathbf{n}$. $\mathsf{Acceleration}$ returns a ω-marking $\mathbf{m}''$ such that $\mathbf{p}^T \cdot \mathbf{m}'' = \mathbf{p}^T \cdot \mathbf{m}_0$. Indeed, since $\mathbf{p}^T \cdot \mathbf{m}' = \mathbf{p}^T \cdot \mathbf{m}_0$, by induction hypothesis, and $\mathbf{p}^T \cdot \mathbf{n} = \mathbf{p}^T \cdot \mathbf{m}_0$, we have $\mathbf{p}^T \cdot \mathbf{m}' = \mathbf{p}^T \cdot \mathbf{n}$. Moreover, $\mathbf{m}' \prec \mathbf{n}$ and hence for all i such that $1 \leq i \leq n$ and $p_i \neq 0$ we have $m'_i = n_i$. We conclude, following Definition 58, that $\mathsf{Acceleration}$ does not put $+\infty$ on those places and $\mathbf{p}^T \cdot \mathbf{m}'' = \mathbf{p}^T \cdot \mathbf{m}_0$.

We conclude that all the ω-markings $\mathbf{m}$ in $fixpoint$ at the end of the algorithm are such that $\mathbf{p}^T \cdot \mathbf{m} = \mathbf{p}^T \cdot \mathbf{m}_0$, hence $dcones(\mathcal{O}_{\mathsf{CS}}(\mathcal{M}, \mathbf{m}_0)) \subseteq dcones(\mathcal{O}(\mathbf{m}_0, \mathbf{p}))$. □

Notice that the MESI protocol example shows that, the reverse, $dcones(\mathcal{O}(\mathbf{m}_0, \mathbf{p})) \subseteq dcones(\mathcal{O}_{\mathsf{CS}}(\mathcal{M}, \mathbf{m}_0))$ is not true in general.

We conclude that $\mathcal{O}_{\mathsf{CS}}(\mathcal{M}, \mathbf{m}_0)$ contains more information than the structural invariants to prune the backward search and so is potentially more effective than the structural invariant heuristic. Instead of opposing the forward and backward approaches, we propose to use them together.

First, if we have to verify that a MTN $\mathcal{M}$, that does not fall in the restrictive class of Multi Isolated Transfer Nets, with the initial marking $\mathbf{m}_0$ cannot reach an upward closed set of Bad markings U, we first compute an over-approximation of the minimal coverability set with the symbolic procedure defined in Section 5.3.1. Then, we check if $dcones(\mathcal{O}_{\mathsf{CS}}(\mathcal{M}, \mathbf{m}_0)) \cap U \neq \emptyset$. If this is the case, then we are done, we know that the MTN cannot reach U. Otherwise, we can use $\mathcal{O}_{\mathsf{CS}}(\mathcal{M}, \mathbf{m}_0)$ in conjunction with the backward search as follow. Instead of computing $\mu X \cdot U \cup \mathsf{pre}(X)$ we compute $\mu X \cdot ((U \sqcap dcones(\mathcal{O}_{\mathsf{CS}}(\mathcal{M}, \mathbf{m}_0))) \cup (\mathsf{pre}(X) \sqcap dcones(\mathcal{O}_{\mathsf{CS}}(\mathcal{M}, \mathbf{m}_0))))$ where $\sqcap$ is defined as : $\sqcap$: upward closed set $\times$ downward closed set $\rightarrow$ upward close set

$$S \sqcap T = \{\mathbf{m} | \exists \mathbf{m}' \in S : \mathbf{m}' \preccurlyeq \mathbf{m} \text{ and } \exists \mathbf{m}'' \in T : \mathbf{m}' \preccurlyeq \mathbf{m}''\}$$

So, we remove from S any upward closed subset that does not intersect with T. This is mainly the same idea that we used in the invariant heuristic defined in Section 5.4.1. But the information computed using the forward search is always at least as precise as the one computed using the structural invariants, and often much more precise.

Using a k-dcCST to represent the forward over-approximation of the reachable markings and k-ucCSTs to represent the sets computed by the backward algorithm, the operation is computed in a similar manner than the minimization operation on k-CSTs (see Section 4.5.7).

However, the computation of such an over-approximation of the reachable markings is theoretically much more difficult than the computation of the structural invariants.

5.5 Experimental Evaluation

In the previous sections, we have presented several algorithms for the backward and forward analysis of MTNs. We have also proposed a heuristic based on structural invariants to prune the backward search space. In this section, we evaluate the practical behavior of the different algorithms and the structural heuristics on several examples of infinite-state MTNs. More precisely, we have considered *mutual exclusion* properties for the parameterized, concurrent production systems like the Mesh 2x2 of [MBC+95, Figure 130, p. 256], its extension to the 3x2 case, and for an extension of the Readers-Writers example given in [Rei86] in which we use several large buffers (e.g. with 45 slots). Furthermore, we have considered *semi-liveness* and *coverability* problems for the PNCSA communication protocol analyzed in [BF99, Fin93]. We also compare our work with alternative approaches.

In Section 5.5.1, we report on results obtained with the exact symbolic algorithms that we propose and an intelligent implementation of the standard enumerative algorithm. In Section 5.5.2, we compare the exact symbolic algorithm with a version where the simulation based heuristics are used. In Section 5.5.3, we show experimental evidences of the benefits of using structural invariants to guide the backward search.

Finally, in Section 5.5.4 and Section 5.5.5 we compare the forward and the backward search together on Petri nets and MTN corresponding to abstraction of JAVA programs.

For the experiments, we used an AMD Athlon 900Mhz with 500Mbytes.

5.5.1 Enumerative versus Symbolic

In the previous sections we have shown that we are able to explore the backward reachable states of an infinite-state MTN starting from an upward-closed set of final markings without enumerating the minimal points of the successive upward-closed sets computed in the fixpoint computation. We have given an algorithm that manipulates directly the graph structure of the k-ucCSTs representing compactly those minimal points.

We have also seen that some problems on k-ucCSTs, related to the symbolic manipulation of minimal elements of an upward-closed set, have a relatively high theoretical complexity (coNP-completeness for example). We show in this subsection that, despite the theoretical complexities of the symbolic operations required for the backward search, the k-ucCSTs based algorithm usually outperforms a good implementation of the enumerative algorithm. Note that the term *enumerative* can be misleading here: only the finite set of minimal elements of the infinite upward-closed sets are enumerated, not all the elements of those sets, of course.

Before presenting our results, we need to be more specific about the enumerative version of the algorithm that we have used. Algorithm 26 is a high level version of the algorithm that we have used (compare it with Algorithm 11 where the exact algorithms (not the simulation-based approximation) are used to compute the normal form and the minimization).

Algorithm 26: $\mathsf{Enumerative}_{\mathsf{BS}}$

Input: $\mathbf{S}$: k-ucCST, $\mathbf{m}_0$: *marking*
Result: *boolean*
begin
  $\mathbf{F} \leftarrow \mathbf{S}$; (*Frontier*);
  $\mathbf{T} \leftarrow \emptyset$; (*Already visited*) ;
  while $\mathbf{F} \neq \emptyset$ **do**
    $\mathbf{T} \leftarrow union(\mathbf{T}, \mathbf{F})$;
    $\mathbf{R} \leftarrow \emptyset$;
    foreach $\mathbf{s} \in elem(\mathbf{F})$ **do**
      foreach *transition* t **do**
        $\mathbf{s}' \leftarrow Pre_{min}(t, \mathbf{s})$;
        if $\nexists \mathbf{s}'' \in elem(\mathbf{T}) \cup elem(\mathbf{S}) \cup elem(\mathbf{R})$ *such that* $\mathbf{s}'' \preccurlyeq \mathbf{s}'$ **then**
          $elem(\mathbf{R}) \leftarrow elem(\mathbf{R}) \cup \{\mathbf{s}'\}$;
    $\mathbf{F} \leftarrow \mathbf{R}$;
  if $\exists \mathbf{s} \in elem(\mathbf{T}) : \mathbf{s} \preccurlyeq \mathbf{m_0}$ **then**
    return *true* ;
  else
    return *false*;
end

The sets $\mathbf{R}$, $\mathbf{F}$, and $\mathbf{T}$ are maintained as k-ucCSTs. We do that in order to isolate as much as possible the effect of enumerating the elements of the frontier instead of manipulating these sets of minimal elements symbolically. Note that, this way, the union operation and the tests $\neg\exists s'' \in \mathbf{T} \cup \mathbf{S} \cup \mathbf{R}$ s.t. $s'' \leq s'$ and $\exists s \in elem(\mathbf{T})$ s.t. $s \preccurlyeq \mathbf{m_0}$ can be done efficiently using the union algorithm on k-sharing trees described in [ZC95] and the algorithm given in the proof of Theorem 14. So, the enumerative search is not naive and the comparison with the symbolic one is fair.

The Symbolic algorithm we used in our experiments in Algorithm 12.

Case-study	P	T	VP	EX	NI	MAX-E	MAX-N	RM	NE	NN	RN	Mem
Mesh2x2$_{k\geq 1}$	32	32	ME	1s	15	53	323	19.2	427	344	2.5	1.6MB
Mesh3x2$_{k\geq 1}$	52	54	ME	18.8s	21	332	1253	7.6	2224	987	0.8	2MB
PNCSA$_1$	31	36	SL	35.3	17	540	1967	11.7	89	176	6.7	2MB
PNCSA$_2$	31	36	C	>4h	11$*$	5042	11323	7.2	–	–	–	–
R/W$_{k\geq 1}$	24	22	ME	>4h	12$*$	15449	5996	1.6	–	–	–	–

Table 5.4: Experiments using the enumerative algorithm: P = Number of places; T = Number of transitions; VP = Verification problem: ME = Mutual exclusion property, C=Covering for a random marking, SL=Semi-liveness; EX = Execution time (h=hour,m=minute,s= second); NI = N. of iterations before termination (with *=before stopping the execution); Max-E(N) = N. of *elements* (*nodes*) of the biggest k-ucCST associated with **S** of Figure 11; RM=MAX-N/(MAX-E $\times$ P) in pct.; NE(N)=Number of *elements* (*nodes*) of the k-ucCST for the fixpoint; RN=NN/(NE $\times$ P) in pct; mem=memory used to compute the fixpoint.

Case-study	P	T	VP	EX	NI	MAX-E	MAX-N	RM	NE	NN	RN	Mem
Mesh2x2$_{K\geq 1}$	32	32	ME	0.8	15	53	323	19	427	344	2.5	2MB
Mesh3x2$_{K\geq 1}$	52	54	ME	6.5	21	332	1253	7.3	2224	987	0.8	3.5MB
PNCSA$_1$	31	36	SL	17.65	17	540	2100	12.5	89	176	6.7	3.7MB
PNCSA$_1$	31	36	C	3h50m	38	8545	17468	6.6	132	368	9	42MB
R/W$_{K\geq 1}$	24	22	ME	>4h	19*	205102	27112	0.5	–	–	–	–

Table 5.5: Experiments using the symbolic algorithm keeping the k-ucCSTs in normal form.

Experimental results obtained with the enumerative algorithm and the symbolic algorithm are shown, respectively, in Table 5.4 and Table 5.5.

For the smaller examples, the enumerative search has execution times that are from two to three times larger than the symbolic search. Note that the largest frontier as well as the fixpoint have the same sizes in the two methods. This is not surprising as the two methods maintain the intermediate results as k-ucCSTs. For the largest examples, the symbolic method works far better. In fact, for the second property of the PNCSA example, the symbolic method is able to compute the 38 iterations needed to reach the fixpoint, whereas the enumerative methods is only able to compute 11 iterations. The advantage of working symbolically is also apparent in the R/W example where the symbolic method is able to compute in 4 hours 19 iterations of the fixpoint computation and manage a frontier of up to 200000 elements when the enumerative algorithm is only able to compute 11 iterations in the same time and seems to be limited to manage frontiers of less than 10000 minimal elements in reasonable time. So, working symbolically seems to be the right way to go.

The other columns in Table 5.4 and Table 5.5 give the following information. We report the size of the set of minimal elements (Max-E) represented by the k-ucCST describing the frontier with the biggest number of nodes (MAX-N) and the ratio (RM) allows us to compare the number of nodes of the k-ucCST with the number of nodes that we would have if there were no sharing. So, the ratio allows us to measure the sharing of nodes between paths of the k-ucCSTs. We report the same information for the k-ucCST corresponding to the fixpoint, i.e. NE is the size of the set it represents, NN is the number of its nodes and RN is the ratio. mem gives the memory used to compute the fixpoint.

In the next two subsections, we will examine the practical behavior of two heuristics that can potentially further improve the efficiency of the symbolic algorithm.

Case-study	P	T	VP	EX	NI	MAX-E	MAX-N	RM	NE	NN	RN	Mem
Mesh2x2$_{k\geq 1}$	32	32	ME	1s	15	54	319	18.4	432	338	2.4	2MB
Mesh3x2$_{k\geq 1}$	52	54	ME	13s	21	334	1198	6.9	2444	1088	0.9	4MB
PNCSA$_1$	31	36	SL	30s	17	548	1972	11.6	89	176	6.4	5MB
PNCSA$_2$	31	36	C	>4h	15∗	23700	29120	4	–	–	–	–
R/W$_{k\geq 1}$	24	22	ME	>4h	16∗	129086	24037	0.8	–	–	–	–

Table 5.6: Experiments using the symbolic algorithm using the simulation relations to approximate the normal form of the k-ucCSTs.

5.5.2 Simulation-based vs Exact Minimization

As discussed in Chapter 4, several basic operations of the *exact* backward symbolic search cannot be done in polynomial time (unless P=NP). However we have proposed exact algorithms that are symbolic and only exponential in the worst-case. The exponential behavior can only occur when the underlying automaton structure of the k-ucCST is made deterministic. Experiments should determine if this pathological behavior often occurs in practice. Because of this hardness result, we have proposed polynomial approximation methods based on the notion of simulation. When using the simulation heuristics, the structure remains deterministic and no determinization is necessary. But the simulation based algorithms are not exact in the sense that they do not compute exactly the normal form and the simplification operation and, even if the sets computed in the fixpoint at all iterations are the same as in the exact method, the k-ucCSTs that represent those sets may contain some redundancy, i.e. the k-ucCSTs may contain elements that are not minimal ones. We compare here the practical behavior of the two approaches: the exact method never generates redundant elements but use algorithms with exponential worst-case behavior while the simulation based heuristic use polynomial time algorithms (with the exception of the fixpoint detection algorithm) but may generate some redundancy in the k-ucCSTs. The first approach corresponds to Algorithm 12 and the second approach corresponds to Algorithm 11.

Experiments with the simulation-based algorithms are reported in Table 5.6. In the examples that we have tested so far, the exponential worst-case behavior of the exact algorithm does not occur and the performances of the exact algorithm are better than the performances of the simulation based algorithm (compare the column Ex of Table 5.5 and Table 5.6). In the small examples, we can see that the heuristic does not generate too much redundancy, in fact, k-ucCSTs of the maximal frontiers as well as the ones of the fixpoints only contain a small number of redundant elements (difference between the number of elements in the simulation based search and the exact search in the tables of Figure 5.5 and Figure 5.6). The simulation based algorithm should not be judged useless, however. In fact, the exact algorithm may have a pathological behavior on some problems and it could be worth to test the simulation based algorithm in those cases.

5.5.3 Practical Evaluation of the Structural Heuristics.

In section 5.4.1, we have proposed to use structural invariants to prune the backward search space. We show here that this heuristic is very effective in practice and allows us to verify systems much more efficiently and more importantly systems that are out of reach of the method without this heuristic.

In all the examples, the use of the structural heuristic improves dramatically the perfor-

Case-study	P	T	VP	EX	NI	MAX-E	MAX-N	RM	NE	NN	RN	Mem
Mesh2x2$_{K\geq1}$	32	32	ME	0.8s	15	53	323	19	424	340	2.5	2.5MB
Mesh3x2$_{K\geq1}$	52	54	ME	6.5s	21	326	1231	7.3	2209	985	1	5MB
PNCSA$_1$	31	36	SL	2.9s	17	85	510	19.3	89	176	6.4	2.5MB
PNCSA$_2$	31	36	C	43s	38	690	1875	8.8	132	368	9	7.5MB
R/W$_{K\geq1}$	24	22	ME	9m 30s	118	16402	3842	1	29119	1936	0.3	34MB

Table 5.7: Experiments using the symbolic algorithm keeping the k-ucCSTs in normal form and enriched with the structural heuristics.

mance of the exact symbolic backward search. The covering property for the PNCSA is solved using the heuristic within $43.09s$ instead of $3h50m$ (that is around 320 times faster). The gain when verifying the R/W example is even clearer. The exact symbolic algorithm without the invariant heuristic is only able to compute the first 19 iterations of the fixpoint computation in 4 hours when the 118 iterations necessary to reach the fixpoint are computed in less than 10 minutes when the structural heuristic is used.

5.5.4 Practical comparison of the backward and forward search on Petri nets

We have applied our symbolic forward and backward algorithms on a set of parameterized Petri nets to compare the practical performances of the two approaches. For these experiments, we used exact algorithms to remove redundancy (and for simplification of k-CSTs), and the polynomial algorithm that implements the heuristic based on place invariants, i.e. Algorithm 23 that does not remove all the possible paths. The results of the experiments are shown in Table 5.8. We have run the backward algorithm with and without the invariant heuristic.

Our set of examples for these experiments is composed of some concurrent and production systems like the multipoll ([MCC97]), the mesh2x2 ([MBC$^+$95]) and its extension to the 3x2 case, the flexible manufacturing system (FMS) of [CM97], the central server model (CSM) of [MBC$^+$95] and the PNCSA protocol analysed in [BF99, Fin93].

As we can see from Table 5.8, the forward search is always faster than the backward search. This is due to the fact that the behaviour of the Petri net is much more regular when executed from its initial markings than when executed backwardly from a possible non reachable set of markings.

As we can see for the PNCSA$_1$ and PNCSA$_2$ examples, where we verify two different properties on the same system, efficiency of the backward search can depend a lot on the property that we want to verify. The invariant heuristic is very useful to obtain reasonable execution times in the backward search, this confirm the intuition that getting some information about (an over-approximation of) the reachable markings is important.

5.5.5 Practical evaluation for the verification of abstractions of Java programs

As stated in Corollary 13, the forward approach cannot be extended to the full class of MTNs. But, as shown in Table 5.8, the backward approach seems to be more *explosion prone* and seems to be useful only when some information about potential reachable markings can be used to *guide* the search. Structural invariants have been used in our experiments on Petri

CS	P	T	$\mathsf{I_{post}}$	$\mathsf{EX_{post}}$	$\mathsf{M_{post}}$	$\mathsf{I_{pre}}$	$\mathsf{EX^1_{pre}}$	$\mathsf{M_{pre}}$	$\mathsf{EX^2_{pre}}$
PNCSA_1	31	36	56	1s	2.3MB	17	17.6s	3.7MB	2.8s
PNCSA_2	31	36	56	1s	2.3MB	38	3h50	42MB	43s
CSM	14	13	10	0.02s	1.4MB	11	0.1s	2.1MB	0.07s
FMS	22	20	16	0.08s	1.6MB	46	6.1s	5.6MB	6s
mesh2x2	32	32	8	0.13s	1.7MB	15	0.8s	2MB	0.8s
mesh3x2	52	54	10	0.48s	2.1MB	21	6.5s	3.5MB	6.5s
multipoll	18	21	11	0.14s	1.6MB	18	2.1s	2.5MB	-

Table 5.8: Comparison between the forward and backward symbolic algorithm on Petri nets: P=No. of places; T = No. of transitions; $\mathsf{I_{post}}$ ($\mathsf{I_{pre}}$) =No. of iterations of the forward (backward) algorithm to reach the fixpoint; $\mathsf{EX_{post}}(\mathsf{EX^1_{pre}})$ = Execution time to reach the forward (backward) fixpoint ; $\mathsf{EX^2_{pre}}$ = Execution time to reach the backward fixpoint using structural invariants (- means that no invariant can be used); $\mathsf{M_{post}}(\mathsf{M_{pre}})$ = Memory usage to reach the forward (backward) fixpoint.

nets to prune the backward search space.

In this section, we report experiments on the backward algorithm applied on MTNs describing abstractions of JAVA programs. Those MTNs have been obtained by applying on JAVA programs the abstraction techniques presented in Chapter 3 and Section 2.3. The symbolic backward algorithm used in those experiments computes the normal form and the minimization of k-ucCSTs, i.e. it does not use the simulation-based heuristics, and uses Algorithm 23 to prune k-ucCST with respect to the structural invariants. For our experiments, we considered the following JAVA programs. The I/D example is a JAVA program composed of many threads manipulating a point object defined by the class of Figure 3.5. The P/C is a classical producer/consumer program, $\mathrm{P/C}_b$ is an extension of the P/C programs where one new kind of producer and consumer is added and introduces a malicious mistake in the program. $\mathrm{P/C}_c$ is the corrected version of $\mathrm{P/C}_b$. The dl, trans and del examples correspond, respectively, to the account management example, the transaction system example and the bounded buffer example of Section 3.3.2. (del_1 and del_2 correspond to two different properties). The conf example is an abstraction of the program presented in [Lea00, pages 203-206] (as for the del example, Conf_1 and Conf_2 correspond to two different properties) that corresponds to a programs composed of threads accessing a shared object with a particular policy. Finally, the queue example [OW99, page 213] is an extension of the dl example where the TSBoolean class manages a FIFO queue of waiting threads.

Table 5.9 shows the results that we obtain by applying our backward symbolic algorithm on our set of examples. Without information to prune the search space, backward algorithm can take a lot of time ($\mathsf{Ex^1_{pre}}$) as already noticed in the case of Basic Petri nets (see the previous section). For instance, many experimentations have been stopped after nine hours of execution.

In those particular examples, the over-approximation computed by the structural invariants is very effective, (see the column $\mathsf{EX^2_{pre}}$ in Table 5.9) but this is not always the case. For example, the MTN of the MESI protocol shown in Figure 5.17 (where dashed arrows represents transfers) has no useful structural invariants: see the description of the example. In those cases, a better over-approximation could be computed by using a forward algorithm, as stated in Section 5.4.2.

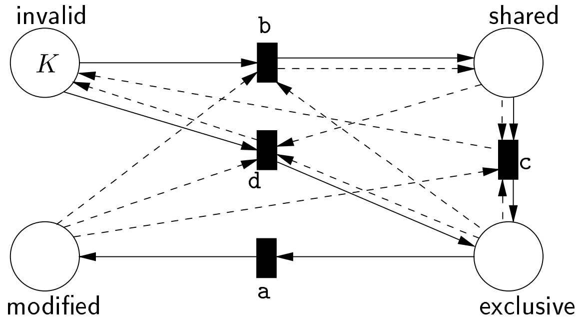

Figure 5.17: MTN of the MESI protocol.

Case Study	P	T	I_{pre}	EX^1_{pre}	N_{pre}	EX^2_{pre}
I/D	32	28	30	20.11s	6MB	3.3s
P/C^1_b	44	37	29	1h12m	33MB	11.45s
P/C^1_c	44	38	29	56m	31MB	0.02s
P/C^2	20	16	19	3.1s	3MB	0.04s
dl	48	42	≥ 38	> 9h	221MB	16s
del_1	50	52	≥ 24	> 9h	179MB	220s
del_2	50	52	≥ 12	> 9h	213MB	3.8s
$conf_1$	30	33	30	45m	19MB	1.7s
$conf_2$	30	33	29	376s	10MB	4.1s
queue	80	104	≥ 12	> 9h	114MB	1h01m
trans	90	117	22	1h12m	18MB	0.54s

Table 5.9: Results obtained by applying the backward symbolic algorithm on MTN.

Notice that some of our examples (I/D, P/C^1_b,P/C^1_c,$conf_1$,$conf_2$,queue,trans) fall into the Multi Isolated Transfer Net[8] subclass but more complex models, for instance the other examples that we consider here, and for which it cannot exist an exact forward search, can be analysed by first computing a forward over-approximation and then use it to prune the backward search.

[8]We could have applied the algorithm of Section 5.3.1 to compute a WCS that is sufficient to verify the safety properties but our remarks in this section are more general and apply to the full class of MTNs.

Chapter 6

Conclusions

6.1 Summary

The central topic of this thesis is the efficient verification of infinite models for concurrent programs, and more precisely multi-threaded JAVA programs, with an unknown (and potentially unbounded) number of threads.

Two major difficulties to solve when we want to verify concurrent programs are:

- the construction of an abstract model;
- the efficient verification of the abstract models.

The first problem is solved by applying the predicate abstraction and the counting abstraction. Applying the predicate abstraction on a program manipulating variables over infinite domains leads to the construction of a boolean program, i.e. a program manipulating only boolean variables, that reflects all the possible behaviours of the real program. Counting abstraction allows us to construct a model amenable to verification where the number of instances of threads is unknown (and potentially unbounded).

The abstract model used in this thesis is based on the Petri net model. As the threads of JAVA programs can use complex primitives of communication that cannot be faithfully described with Petri nets, we added some extensions to the basic model of Petri nets. More precisely, the abstract models that we consider, called multi transfer nets, are Petri nets extended with unbounded and bounded transfers. Unbounded transfers correspond to the classical extension of Petri nets (see [Cia94]) and allow us to encode the behaviour of the notifyAll primitive of JAVA. Bounded transfers is a new extension of Petri nets that allows us to describe the behaviour of the notify primitive of JAVA.

An interesting topic of further research would be to search for the more general models that can be automatically verified. We studied our new extension with this goal in mind. Unfortunately, most of the properties that are decidable on Petri nets become undecidable when Petri nets are extended with bounded transfers. Notice that for those properties, semi-algorithms could be defined, and hence we could have (semi) algorithmic methods that could be useful in practice.

However, the coverability problem, corresponding the reachability of upward closed sets of markings, remains decidable. This is a positive result because it allows us to automatically verify a large class of interesting properties, including for example, mutual exclusion properties.

Verification of the coverability problem on multi transfer nets is achieved using algorithms that manipulate upward closed and downward closed sets of markings. Such (potentially) infinite sets can be finitely described through their finite sets of generators. So, algorithms manipulate upward and downward closed sets through their finite sets of generators. However, even if the generator sets are finite, they can become very large when we analyse models. This is an important motivation to define data structures that efficiently represent those sets.

BDD-like data structures are classical efficient representations of finite sets of tuples. Starting from the BDD-like data structure of D. Zampuniéris, called Sharing Trees, we extended them to allow us to represent upward closed and downward closed sets of tuples over naturals.

Our data structure, called Covering Sharing Trees, is generic (as Sharing Trees) in the sense that only few assumptions are given on the domain of tuples. It follows that several kinds of datas can be encoded using our Covering Sharing Trees. For instance, upward closed and downward closed sets of tuples over strings can be represented with Covering Sharing Trees. Such sets are interesting because they are manipulated by model checking algorithms used, for instance, to verify Lossy FIFO channels models.

It is important to note that all the results on Covering Sharing Trees are independent from the value domains used and hold, for instance, both for (upward and downward) closed sets of tuples over naturals and (upward and downward) closed sets of tuples over strings.

Symbolic forward and backward algorithms have been defined to verify covering properties on multi transfer nets. They are symbolic in the sense that they do not manipulate sets of generators enumeratively, but use Covering Sharing Trees to represent and manipulate sets of generator without enumerating their elements. The forward algorithm starts from the initial markings and tries to reach an upward closed set of markings by computing the successor markings of the initial ones. The markings of the upward closed set often correspond to bad configurations of the system to verify. The backward algorithm makes the converse. This means that it starts from an upward closed set of markings and tries to reach an initial marking by computing the predecessor markings of the upward closed set.

Practical evaluations of those algorithms on Petri nets and abstractions of JAVA programs show the efficiency of the forward approach. However, this algorithm cannot be generalised to the full class of multi transfer nets. On the other hand, practical evaluations show that backward algorithm is less efficient than the forward one but can be applied on the full class of multi transfer nets. To make the backward algorithm also useful in practice, some heuristics must be defined. We proposed the use of structural invariants to define over-approximations of the reachable state space of multi transfer nets from a set of initial markings. These over-approximations are used to guide the backward search to the initial markings.

When the forward approach is applied on the full class of multi transfer nets, the algorithm computes an over-approximation of the reachable state space. This over-approximation can be also used to guide the backward search.

Finally, practical evaluations of the extended backward algorithm on abstractions of JAVA programs show that backward approach can be efficient when some information is given to guide the search.

6.2 Future works

The work presented in this thesis can be extended in several ways. We present here a non exhaustive enumeration of possible extensions.

1. To fully automate the verification of JAVA programs with an unbounded number of threads, an algorithm that would allow us to construct concurrent boolean programs from JAVA programs must be defined. To reach this goal, several problems must be solved.

   First, a methodology of abstraction must be formally defined. For instance, a problem to fix is the way to keep some information on the identity of threads. Counting abstraction loose such an information. But keeping the identity of a finite number of threads can be achieved by abstracting these threads in a particular way (see examples of Section 1.3.2).

   Second, when models are generated automatically they are often large, even for small systems, and some algorithms must be applied to make them compact. As the efficiency of the verification methods depends on the size of the analysed models, it should be interesting to define several techniques to generate abstract models as small as possible. So, particular algorithms could be defined to optimize the models generated at each step of the abstraction phase.

2. Another work that could be initiated is the extension of the class of sets manipulated by model checking algorithms. More precisely, each node of upward closed covering sharing trees and downward closed covering sharing trees encodes respectively constraints of the form $x \geq c$ and $x \leq c$. A natural extension could be to associate constraints of the form $x \in [c, d]$ to nodes.

   Such sets would allow us to verify more general abstract models. For examples, we could add tests to zero on places of multi transfer nets allowing us to construct more precise models to analyse. With such an extension, all the interesting properties to verify on extended Petri nets become undecidable.

   As a second consequence of extending the class of sets manipulated by algorithms, we mention that the class of properties to verify can be extended. More precisely, we could check the reachability of sets that are more general than upward closed sets of markings. For instance, this would allow us to decide general deadlock properties.

3. As we already mentioned it, Covering Sharing Tree can be used to encode downward and upward closed set of tuples over strings. Algorithm for the verification of Lossy FIFO channels models using Covering Sharing Trees can be defined and evaluated on practical examples.

   This is not an easy work as forward algorithms for Lossy FIFO channels are not guaranteed to terminate in theory. To overcome this problem, acceleration techniques have been proposed to ensure termination in practice (see [ABJ98]). Those acceleration techniques working symbolically on Covering Sharing Trees should be then defined to ensure termination in practice.

4. As we showed it during the practical evaluation of the verification algorithms on the multi transfer nets, it is crucial to give some information to backward algorithms to guide them during the backward search. We proposed to use structural invariants of nets and the result of forward algorithms to give an over-approximation of the reachable state space to the backward search. It should be interesting to have another kinds of information to guide the backward search. This is also a topic of further research.

Bibliography

[AAB00] A. Annichini, E. Asarin, and A. Bouajjani. Symbolic techniques for parametric reasoning about counter and clock systems. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 419–434. Springer, 2000.

[ABJ98] P.A. Abdulla, A Bouajjani, and B Jonsson. On-the-Fly Analysis of Systems with Unbounded, Lossy FIFO Channels. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 305–318. Springer, 1998.

[ABJN99] P.A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilson. Handling global conditions in parameterized system verification. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 134–145. Springer, 1999.

[ABS01] A. Annichini, A. Bouajjani, and M. Sighireanu. TReX : A tool for reachabillity analysis of complex systems. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 368–372. Springer, 2001.

[ACJT96] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General Decidability Theorems for Infinite-state Systems. In *Proceedings of the 11th Annual Symposium on Logic in Comuter Science (LICS'96)*, pages 313–321. IEEE Computer Society Press, 1996.

[AIN00] P.A. Abdulla, P. Iyer, and A. Nylén. Unfoldings of Unbounded Petri Nets. In *Proceedings of the 12th International Conference on Computer-Aided Verification Conference (CAV'00)*, volume 1855 of *LNCS*, pages 495–507. Springer, 2000.

[AJ93] P.A. Abdulla and B. Jonsson. Verifying Programs with Unreliable Channels. In *Proceedings of the 8th IEEE International Symposium in Logic in Computer Science (LICS'93)*, pages 160–170. IEEE Computer Society Press, 1993.

[AJNd02] P.A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Regular Model Checking Made Simple and Efficient. In *In Proceedings ot the 13th International Conference on Concurrency Theory (CONCUR'02)*, volume 2421 of *LNCS*, pages 116–130. Springer, 2002.

[AN01] P.A. Abdulla and A. Nylén. Timed Petri Nets and BQOs. In *Proceedings of the 22nd International Conference on Application and Theory of Petri nets (ICATPN'01)*, volume 2075 of *LNCS*, pages 53–70. Springer, 2001.

[AO97] K. R. Apt and E.-R. Oldedrog. *Verification of Sequential and Concurrent Programs, second edition.* Springer-Verlag, 1997.

[Bar96] J. Barnes. *Programming in ADA 95.* Addison-Wesley, 1996.

[BC96] A. Boudet and H. Common. Diophantine Equations, Presburger Arithmetic And Finite Automata. In *Proceedings of the 21st International Colloquium on Trees in Algebra and Programming (CAAP'96)*, volume 1059 of *LNCS*, pages 30–43. Springer, 1996.

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science (LICS'90)*, pages 428–439. IEEE Computer Society Press, 1990.

[BCR01] T. Ball, S. Chaki, and S.K. Rajamani. Parameterized Verification of Multithreaded Software Libraries. In *Proceedings of the Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 158–173. Springer, 2001.

[BD02] M. Bozzano and G. Delzanno. Automated Protocol Verification in Linear Logic. In *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, pages 295–308. Springer, 2002.

[BF99] B. Bérard and L. Fribourg. Reachability Analysis of (Timed) Petri Nets Using Real Arithmetic. In *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'99)*, volume 1664 of *LNCS*, pages 178–193. Springer, 1999.

[BFLP03] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: Fast acceleration of symbolic transition systems. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, will be published in LNCS. Springer, 2003.

[BG96] B. Boigelot and P. Godefroid. Symbolic Verification of Communication Protocols with Infinite State Spaces using QDDs. In *Proceedings of the 8th International Conference on Computer-Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 1–12. Springer, 1996.

[BGP97] T. Bultan, R. Gerber, and W. Pugh. Symbolic Model Checking of Infinite-state Systems using Presburger Arithmetics. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 400–411, 1997.

[BHPV00] G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder second generation of a java model checker. In *Proceedings of the Post-CAV Workshop on Advances in Verification 2000 (WAVE 2000)*, 2000.

[Bil91] J. Billington. *Extensions to Coloured Petri nets and their applications to Protocols.* PhD thesis, University of Cambridge, 1991.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In *Proceedings of the 2th International Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 403–418. Springer, 2000.

[BMMR01] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of The 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2001)*, pages 203–213. ACM Press, 2001.

[BMR02] T. Ball, T. Millstein, and S.K. Rajamani. Polymorphic predicate abstraction. Technical Report MSR-TR-2001-10, Microsoft Research, 2002.

[Boi99] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Université de Liège, 1999.

[BPR02] T. Ball, A. Podelski, and S.K. Rajamani. Relative Completeness of Abstraction Refinement for Software Model Checking. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *LNCS*, pages 158–172. Springer, 2002.

[BR00] T. Ball and S.K. Rajamani. Boolean programs: A model and process for software analysis. Technical report, Microsoft Research, 2000.

[Bry86] R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transaction on Computers*, C-35(8):667–691, 1986.

[BSV94] F. Balarin and A.L. Sangiovanni-Vincentelli. On The Automatic Computation Of Network Invariants. In *Proceedings of the 6th International Conference on Computer-Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 234–246. springer, 1994.

[But97] R. Butenhof. *Programming With POSIX Threads*. Addison-Wesley, 1997.

[BW94] B. Boigelot and P. Wolper. Symbolic Verification with Periodic Sets. In *Proceedings of the 6th Conference on Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 55–67. Springer, 1994.

[BW98] B. Boigelot and P. Wolper. Verifying Systems with Infinite but Regular State Space. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 88–97. Springer, 1998.

[CFGR95] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation*, 24(1-2):47–68, 1995.

[Che00] J. Chen. On verifying distributed multithreaded java programs. In *Proceedings of the 33th IEEE Hawaii International Conference on System Sciences (HICSS-33)*. IEEE Computer Society Press, 2000.

[Cia94] G. Ciardo. Petri nets with marking-dependent arc multiplicity: properties and analysis. In *Proceedings of the 15th International Conference on Applications* *and Theory of Petri Nets (ICATPN 94)*, volume 815 of *LNCS*, pages 179–198. Springer, 1994.

[CM97] G. Ciardo and A.S. Miner. Storage Alternatives for Large Structured State Space. In *Proceedings of the 9th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (Tools'97)*, volume 1245 of *LNCS*, pages 44–57. Springer, 1997.

[DB01] G. Delzanno and T. Bultan. Constraint-based Verification of Client-server Protocols. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP2001)*, volume 2239 of *LNCS*, pages 286–301. Srpinger, 2001.

[DDP99] S. Das, D.L. Dill, and S. Park. Experience with predicate abstraction. In *Proceedings of the 11th Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 160–171. Springer, 1999.

[Del00] G. Delzanno. Automatic Verification of Parameterized Cache Coherence Protocols. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 53–68. Springer, 2000.

[DEP99] G. Delzanno, J. Esparza, and A. Podelski. Constraint-based Analysis of Broadcast Protocols. In *Proceedings of the 13th Annual Conference of the European Association for Computer Science Logic (CSL'99)*, volume 1683 of *LNCS*, pages 50–66. Springer, 1999.

[DFS98] C. Dufourd, A. Finkel, and Ph. Schnoebelen. Reset Nets Between Decidability and Undecidability. In *In Proceedings of the 25th International Colloquium on Automata, Languages, and Programming (ICALP'98)*, volume 1443 of *LNCS*, pages 103–115. Springer, 1998.

[DIS98] C. Demartini, R. Iosif, and R. Sisto. Modeling and Validation of Java Multithreading Applications using SPIN. In *Model Checking Software, Proceedings of the 4th International SPIN Workshop (SPIN98)*, 1998.

[DLS01] D. Dams, Y. Lakhnech, and M. Steffen. Iterating Transducer. In *Proceeding of the 13th International Conference of Computer-Aided Verification (CAV 01)*, volume 2102 of *LNCS*, pages 286–297. Springer, 2001.

[DR00] G. Delzanno and J.-F. Raskin. Symbolic Representation of Upward Closed Sets. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, volume 1785 of *LNCS*, pages 426–440. Springer, 2000.

[DRB01] G. Delzanno, J.-F. Raskin, and L. Van Begin. Attacking Symbolic State Explosion. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *LNCS*, pages 298–310. Springer, 2001.

[DRB02] G. Delzanno, J.-F. Raskin, and L. Van Begin. Covering Sharing Trees: Efficient Data Structurs for the Automated Verification of Parametrized Systems. *accepted for publication in Software Tools for Technology Transfer Manuscript*, 2002.

[DRVB02] G. Delzanno, J-F. Raskin, and L. Van Begin. Towards the Automated Verification of Multithreaded Java Programs. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *LNCS*, pages 173–187. Springer, 2002.

[EFM99] J. Esparza, A. Finkel, and R. Mayr. On the Verification of Broadcast Protocols. In *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 352–359. IEEE Computer Society Press, 1999.

[EK00] E.A. Emerson and V. Kahlon. Reducing Model Checking of the Many to the Few. In *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*, volume 1831 of *LNCS*, pages 236–254. Springer, 2000.

[EN98] E. A. Emerson and K. S. Namjoshi. On Model Checking for Non-deterministic Infinite-state Systems. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS '98)*, pages 70–80. IEEE Computer Society Press, 1998.

[Esp94] J. Esparza. On the Decidabilty of Model Checking for Several mu-calculi and Petri Nets. In *Proceedings of the 19th International Colloquium on Trees in Algebra and Programming (CAAP 1994)*, volume 787 of *LNCs*, pages 115–129. Springer, 1994.

[Fin90] A. Finkel. Reduction and Covering of Infinite Reachability Trees. *Information and Computation*, 89(2):144–179, 1990.

[Fin93] A. Finkel. The minimal coverability graph for Petri nets. In *Proceedings of Advances in Petri Nets*, volume 674 of *LNCS*, pages 210–243. Springer, 1993.

[FL02] A. Finkel and J. Leroux. How to Compose Presburger-Accelerations: Applications to Broadcast Protocols. In *Proceedings of the 22th International Conference on the Foundations of Software Technology and Theoretical Computer Science (FST&TCS 2002)*, volume 2556 of *LNCS*, pages 145–156. Springer, 2002.

[Fla97] D. Flanagan. *Java in a Nutshell.* O'reilly & Associates, inc., 1997.

[FP01] D. Frisman and A. Pnueli. Beyond regular model checking. In *Proceedings of the 21st Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2001)*, volume 2245 of *LNCS*, pages 156–170. Springer, 2001.

[FRSVB02] A. Finkel, J.-F. Raskin, M. Samuelides, and L. Van Begin. Monotonic Extensisions of Petri Nets : Forward and Backward Search Revisited. In *Proceedings of the 4th international workshop on verification of infinite-state systems (INFINITY 2002)*, volume 68 of *ENTCS*. Elsevier, 2002.

[FS01] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.

[FWW97] A. Finkel, B. Willems, and P. Wolper. A Direct Symbolic Approach to Model Checking Pushdown systems. *Electronic Notes in Theoretical Computer Science*, 9, 1997.

[Gan02] P. Ganty. Algorithmes et Structures de Données Efficaces Pour la Manipulation de Contraintes Sur les Intervalles. Master's thesis, Université Libre de Bruxelles, 2002.

[Gra97a] B. Grahlmann. The PEP Tool. In *Proceedings of the 9th Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 440–443. Springer, 1997.

[Gra97b] M. Grand. *Java Language Reference*. o'reilly, 1997.

[GS92] S. M. German and A. P. Sistla. Reasoning about Systems with Many Processes. *Journal of ACM*, 39(3):675–735, 1992.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *Proceeding of the 9th Conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[Han93] J. Handy. *The Cache Memory Book*. Academic Press, 1993.

[HD01] J. Hatcliff and M. Dwyer. Using the Bandera Tool Set to Model-check Properties of Concurrent Java Software. In *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR 2001)*, volume 2154 of *LNCS*, pages 39–58. Springer, 2001.

[HHK95] M. R. Henzinger, T. A. Henzinger, and P. K. Kopke. Computing Simulations on Finite and Infinite Graphs. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science (FOCS'95)*, pages 453–462. IEEE Society Press, 1995.

[HHWT97] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a Model Checker for Hybrid Systems. In *Proceedings of the 9th Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 460–463. Springer, 1997.

[HJMS02] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, pages 58–70. ACM Press, 2002.

[ID99] C.N. Ip and D. Dill. Verifying Systems with Replicated Components in Murφ. *Formal Methods in System Design*, 9(1):41–75, 1999.

[Ios00] R. Iosif. *A Formal Approach to Practical Software Verification*. PhD thesis, Politecnico di Torino University, 2000.

[JN00] B. Jonsson and N. Nilsson. Transitive Closures of Regular Relations for Verifying Infinite-State Systems. In *Proceedings of the 6th International Conference on Tools and Algorithms for the construction and Analysis of Systems (TACAS 2000)*, volume 1785 of *LNCS*, pages 220–234. Springer, 2000.

[Joh90] D. S. Johnson. A Catalog of Complexity Classes. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science Volume A : Algorithms and Complexity*, pages 67–161. Elsevier, 1990.

[KM69] R. M. Karp and R. E. Miller. Parallel Program Schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.

[KM89] R. P. Kurshan and K.L. McMillan. A Structural Induction Theorem for Processes. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing (PODC 89)*, pages 239–247. ACM Press, 1989.

[Kos82] S.R. Kosaraju. Decidability of reachability in vector addition systems. In *Proceedings of the 14th ACM Symposium on Theory of Computing (STOC'82)*, pages 267–281. ACM Press, 1982.

[LAS] The Liège Automata-based Symbolic Handler (LASH). available at `http://www.montefiore.ulg.ac.be/ boigelot/research/lash`.

[Lea00] D. Lea. *Concurrent Programming in Java.* The Java Series. Addison Wesley, 2000.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic Verification of Parameterized Linear Networks of Processes. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 346–357. ACM Press, 1997.

[May84] E.W. Mayr. An algorithm for the general petri net reachability problem. *SIAM Journal of Computing*, 3(13):441–460, 1984.

[MBC+95] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. Modelling with Generalized Stochastic Petri Nets. *Series in Parallel Computing*, 1995.

[MCC97] P. Marenzoni, S. Caselli, and G. Conte. Analysis of Large GSPN Models : A Distributed Solution Tool. In *Proceedings of the 6th International Workshop on Petri Nets and Performance (PNPM'97)*, pages 122–131. IEEE Computer Society Press, 1997.

[Min67] N.M. Minsky. *Finite and Infinite Machines.* Englewood Cliffs, N.J., Prentice-Hall, 1967.

[MLAH99] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference Decision Diagrams. In *Proceedings of the 13th Annual Conference of the European Association for Computer Science Logic (CSL'99)*, volume 1683 of *LNCS*, pages 111–125. Springer, 1999.

[OW99] Scott Oaks and H. Wong. *Java Threads (Second Edition).* O'Reilly, 1999.

[PRZ01] A. Pnueli, S. Ruah, and L. D. Zuck. Automatic Deductive Verification with Invisible Invariants. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, volume 2031 of *LNCS*, pages 82–97. Springer, 2001.

[PSSD00] D. Park, U. Stern, J.U. Skakkebaek, and D.L. Dill. Java Model Checking. In *Proceedings of the 1st International Workshop on Automated Program Analysis, Testing, and Verification*, 2000.

[Rei86] W. Reisig. *Petri Nets. An introduction.* Springer, 1986.

[RV02] T. Rybina and A. Voronkov. Using canonical representations of solutions to speed up infinite-state model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 386–400. Springer, 2002.

[RVB03] J.-F. Raskin and L. Van Begin. Petri Nets with Non-blocking Arcs are Difficult to Analyse. In *Proceedings of the 5th International Workshop on Verification of Infinite-state Systems (INFINITY 2003)*, volume 96 of *ENTCS*. Elsevier, 2003.

[SL01] S.D. Stoller and Y.A. Liu. Transformations for model checking distributed java programs. In *Proceedings of the 8th International SPIN Workshop (SPIN 2001)*, volume 2057 of *LNCS*, pages 192–199. Springer, 2001.

[SS99] H. Saidi and N. Shankar. Abstract and model check while you prove. In *Proceedings of the 11th International Conference on Computer-Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 443–454. Springer, 1999.

[ST98] K. Strehl and L. Thiele. Symbolic Model Checking of Process Networks Using Interval Diagram Techniques. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'98)*, pages 686–692, 1998.

[ST99] K. Strehl and L. Thiele. Interval Diagram Techniques for Symbolic Model Checking of Petri Nets. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'99)*, pages 756–757, 1999.

[ST00] K. Strehl and L. Thiele. Interval Diagrams for Efficient Symbolic Verification of Process Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(8):939–956, 2000.

[STC98] M. Silva, E. Teruel, and J. M. Colom. Linear Algebraic and Linear Programming Techniques for Analysis of Place/Transition Net Systems. In *Proceedings of Advances in Petri Nets*, volume 1491 of *LNCS*, pages 308–309. Springer, 1998.

[Sto02] S.D. Stoller. Model-Checking Multi-Threaded Distributed Java Programs. *International Journal on Software Tools for Technology Transfer*, 4(1):71–91, 2002.

[Ter94] E. Teruel. *Structure Theory of Weighted Place/Transition Net Systems: The Equal Conflict Hiatus.* PhD thesis, University of Zaragoza, Zaragoza, Spain, 1994.

[Tou01] T. Touili. Regular model checking using widening techniques. *Electronic Notes in Theoretical Computer Science*, 50(3), 2001.

[Val78] R. Valk. On the computational power of extended petri nets. In *Proceedings of the 7th symposium on Mathematical Foundations of Computer Science*, volume 64 of *LNCS*, pages 527–535. Springer, 1978.

[VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00)*, pages 3–13. IEEE Press, 2000.

[WB95] P. Wolper and B. Boigelot. An Automata-theoretic Approach to Presburger Arithmetic Constraints. In *Proceedings of the 2nd International Symposium on Static Analysis (SAS'95)*, volume 983 of *LNCS*, pages 21–32. Springer, 1995.

[WB00] P. Wolper and B. Boigelot. On the Construction of Automata from linear arithmetic constraints. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, volume 1785 of *LNCS*, pages 1–19. Springer, 2000.

[WL89] P. Wolper and V. Lovinfosse. Verifying Properties of Large sets of processes with network invariants. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 68–80. Springer, 1989.

[Zam97] D. Zampuniéris. *The Sharing Tree Data Structure: Theory and Applicationsin Formal Verification*. PhD thesis, Facultés Universitaires Notre-Dame de la Paix, Namur, Belgium, 1997.

[ZC95] D. Zampuniéris and B. Le Charlier. Efficient Handling of Large Sets of Tuples with Sharing Trees. In *Proceedings of the Data Compressions Conference (DCC'95)*, page 428. IEEE Computer Society Press, 1995.

List of Figures

List of Tables

List of Algorithms

Appendix

.1 Algorithms for adjusting conditions 5 and 6 of Definition 42

Algorithm 27: Adjust_Condition5

Input: $\mathbf{S}$: pre-sharing tree

begin

  foreach $i : 0$ *to* k (*top to bottom*) **do**

1   **foreach** *node* n *of* $N_i^{\mathbf{S}}$ **do**

      $newsucc \leftarrow \emptyset$;

      foreach *value* c *such that* $\exists n' \in succ^{\mathbf{S}}(n) : val^{\mathbf{S}}(n') = c$ **do**

        Construct a new node n'' *with* $val^{\mathbf{S}}(n'') = c$;

        foreach $n' \in succ^{\mathbf{S}}(n)$ *such that* $val^{\mathbf{S}}(n') = c$ **do**

          $succ^{\mathbf{S}}(n'') \leftarrow succ^{\mathbf{S}}(n'') \cup succ^{\mathbf{S}}(n')$;

        if $\exists n' \in N_{i+1}^{\mathbf{S}} : val^{\mathbf{S}}(n') = c \wedge SameSons(n', n'')$ **then**

          let m *be such a node and* $newsucc \leftarrow newsucc \cup \{m\}$;

        else

          $newsucc \leftarrow newsucc \cup \{n''\}$;

          $N_{i+1}^{\mathbf{S}} \leftarrow N_{i+1}^{\mathbf{S}} \cup \{n''\}$;

      $succ^{\mathbf{S}}(n) \leftarrow newsucc$;

2   **foreach** *node* $n \in N_{i+1}^{\mathbf{S}}$ **do**

      if $\nexists n' \in N_i^{\mathbf{S}} : n \in succ^{\mathbf{S}}(n')$ **then**

        remove n *from* $N_{i+1}^{\mathbf{S}}$;

end

Algorithm 28: Adjust_Condition6

Input: $\mathbf{S}$: pre-sharing tree
begin
  foreach $n \in N^{\mathbf{S}}$ **do**
    $remove(n) \leftarrow false;$
  foreach $i : k+1$ *to* 1 $(bottom - up)$ **do**
    foreach *node* $n \in N_i^{\mathbf{S}}$ *with* $remove(n) = false$ **do**
      foreach *node* $n' \in N_i^{\mathbf{S}} (n' \neq n)$ *such that* $val^{\mathbf{S}}(n) = val^{\mathbf{S}}(n') \wedge SameSons(n, n')$ **do**
        foreach n'' *such that* $n' \in succ^{\mathbf{S}}(n'')$ **do**
          $succ^{\mathbf{S}}(n'') \setminus \{n'\};$
          $succ^{\mathbf{S}}(n'') \cup \{n\};$
        $remove(n') \leftarrow true;$
  foreach $n \in N_i^{\mathbf{S}}$ *such that* $remove(n) = true$ **do**
    $N_i^{\mathbf{S}} \setminus \{n\};$
end

.2 Algorithm of the cartesian product on k-sharing trees

Algorithm 29: Cartesian_Product_First_Dim

Input: $n_{\mathbf{S}_1}$: node; $n_{\mathbf{S}_2}$: node
Result: a node
begin
  if $val^{\mathbf{S}_1}(n_{\mathbf{S}_1}) = \bot$ **then**
    $Sol \leftarrow new\ node\ such\ that\ val^{\mathbf{Prod}}(Sol) = \bot$;
  else
    $Sol \leftarrow new\ node\ such\ that\ val^{\mathbf{Prod}}(Sol) = val^{\mathbf{S}_1}(n_{\mathbf{S}_1})$;
    foreach $s \in succ^{\mathbf{S}_2}(n_{\mathbf{S}_2})$ **do**
      $Son \leftarrow$ memo?$(n_{\mathbf{S}_1}, s)$;
      if $Son = notmemorized$ **then**
        $Son \leftarrow$ Cartesian_Product_Second_Dim$(n_{\mathbf{S}_1}, s)$;
      $succ^{\mathbf{Prod}}(Sol) \leftarrow succ^{\mathbf{Prod}}(Sol) \cup \{Son\}$;
  memo!$(n_{\mathbf{S}_1}, n_{\mathbf{S}_2}, n)$;
  return Sol ;
end

Algorithm 30: Cartesian_Product_Second_Dim

Input: $n_{\mathbf{S}_1}$: node; $n_{\mathbf{S}_2}$: node
Result: a node
begin
  $Sol \leftarrow new\ node\ such\ that\ val^{\mathbf{Prod}}(Sol) = val^{\mathbf{S}_2}(n_{\mathbf{S}_2})$;
  foreach $s \in succ^{\mathbf{S}_1}(n_{\mathbf{S}_1})$ **do**
    $Son \leftarrow$ memo?$(s, n_{\mathbf{S}_2})$;
    if $Son = notmemorized$ **then**
      $Son \leftarrow$ Cartesian_Product_First_Dim$(s, n_{\mathbf{S}_2})$;
    $succ^{\mathbf{Prod}}(Sol) \leftarrow succ^{\mathbf{Prod}}(Sol) \cup \{Son\}$;
  memo!$(n_{\mathbf{S}_1}, n_{\mathbf{S}_2}, n)$;
  return Sol ;
end

Algorithm 31: Cartesian_Product

Input: $\mathbf{S}_1$: k-sharing tree; $\mathbf{S}_2$:k- sharing tree
Result: a k-sharing tree
begin
    Let **Prod** be the k-sharing tree rooted by $root^{\mathbf{Prod}}$ built as follows ;
    $succ^{\mathbf{Prod}}(root^{\mathbf{Prod}}) \leftarrow \emptyset$;
    foreach $s \in succ^{\mathbf{S}_1}(root^{\mathbf{S}_1})$ **do**
        $Son \leftarrow$ Cartesian_Product_First_Dim$(s, root^{\mathbf{S}_2})$;
        $succ^{\mathbf{Prod}}(root^{\mathbf{Prod}}) \leftarrow succ^{\mathbf{Prod}}(root^{\mathbf{Prod}}) \cup \{Son\}$;
    Adjust_Condition6(**Prod**) ;
    return Prod ;
end

.3 Proof of Lemma 7

Proof. We are now about to exhibit a proof by induction on the length of e.

Basis case: If the length of e is 1, then by definition of the initial states of B and G_B, we have $\Phi(e) = Created(R)$.

Induction step: Let us suppose that the lemma holds for every run of length $k \leq i$. Let $e = e' \cdot S \cdot S'$ be a run of B of length $i+1$ with $S = \langle \Gamma, \Lambda, \{s_1, \ldots, s_n\}\rangle$ and $S' = \langle \Gamma', \Lambda', \{s'_1, \ldots, s'_{n'}\}\rangle$, and $R \cdot S_G$ with $S_G = \langle \Gamma \cup f(\Lambda), l_1, \ldots, l_m\rangle$ be an execution of G_B of lenght i such that $\Phi(e' \cdot S) = Created(R \cdot S_G)$. We are now going to show that there exists a successor S'_G of S_G such that $\Phi(S') = Created(S'_G)$, and hence $\Phi(e' \cdot S \cdot S') = Created(R \cdot S_G \cdot S'_G)$.

Depending on S, different rules of Section 3.1.2 can be applied to obtain S' from S. We have to consider different cases:

(a) There exists $1 \leq i \leq n$ such that $s_i = \langle [\ell]u_1, \ldots, u_k := v_1, \ldots, v_k; [\ell']I' \cdot S'', \rho\rangle$. Let us consider a permutation of the variable indexes : $\{1, \ldots k\} \to \{i_1 \ldots i_k\}$ such that there is l with $0 \leq l \leq k$ such that i_j with $1 \leq j \leq l$ is a local variable and i_j with $l+1 \leq j \leq k$ is a global variable.

S' is such that $s'_i = \langle [\ell']I' \cdot S'', \rho[u_{i_1} = v_{i_1} \ldots u_{i_l} = v_{i_l}]\rangle$; (ii) $n = n'$, $\Gamma' = \Gamma[u_{l+1} = v_{l+1} \ldots u_{i_k} = v_{i_k}]$, $\Lambda' = \Lambda$ and (iii) for all $1 \leq j \leq n$ $(j \neq i)$ we have $s'_j = s_j$. We have that $\Phi(S')$ is exactly $\Phi(S)$ where the valuation of the global variables becomes $\Gamma[u_{l+1} = v_{l+1} \ldots u_{i_k} = v_{i_k}] \cup f(\Lambda)$ and one local machine moves from the local state $\langle \rho, \ell\rangle$ to $\langle \rho[u_{i_1} = v_{i_1} \ldots u_{i_l} = v_{i_l}], \ell'\rangle$.

Moreover, as $\Phi(S) = Created(S_G)$ by hypothesis, there exists $1 \leq j \leq m$ such that $l_j = \langle \rho, \ell\rangle$ in S_G. Following Rule 2a and the semantics of global machines, S_G has one successor S'_G that is exactly S_G where the valuation of the global variables becomes $\Gamma[u_{l+1} = v_{l+1} \ldots u_{i_k} = v_{i_k}] \cup f(\Lambda)$ and one local machine moved from $\langle \rho, \ell\rangle$ to $\langle \rho[u_{i_1} = v_{i_1} \ldots u_{i_l} = v_{i_l}], \ell'\rangle$. So, $\Phi(S') = Created(S'_G)$;

(b) There exists $1 \leq i \leq n$ such that $s_i = \langle [\ell]\texttt{if } (B) \texttt{ then } \{[\ell_1]I_1 \cdot S_1\} \texttt{ else } \{[\ell_2]I_2 \cdot S_2\} \cdot S, \rho\rangle$ and

- either $\Gamma \cup \rho \models B$, and S' is such that $s'_i = \langle [\ell_1]I_1 \cdot S_1 \cdot S, \rho\rangle$; (ii) $n = n'$, $\Gamma' = \Gamma$, $\Lambda' = \Lambda$ and (iii) for all $1 \leq j \leq n$ $(j \neq i)$ we have $s'_j = s_j$. We have that $\Phi(S')$ is exactly $\Phi(S)$ where one local machine moves from $\langle \rho, \ell\rangle$ to $\langle \rho, \ell_1\rangle$.

  Moreover, as $\Phi(S) = Created(S_G)$, there exists $1 \leq j \leq m$ such that $l_j = \langle \rho, \ell\rangle$ in S_G and the valuation of the global variable is $\Gamma \cup f(\Lambda)$. Following Rule 2b of section 3.2 and the semantics of global machines, S_G has one successor S'_G that is exactly S_G where one local machine moved from the local state $\langle \rho, \ell\rangle$ to $\langle \rho, \ell_1\rangle$. So, $\Phi(S') = Created(S'_G)$;

- Otherwise $\Gamma \cup \rho \not\models B$, and S' is such that $s'_i = \langle [\ell_2]I_2 \cdot S_2 \cdot S, \rho\rangle$; (ii) $n = n'$, $\Gamma' = \Gamma$, $\Lambda' = \Lambda$ and (iii) for all $1 \leq j \leq n$ $(j \neq i)$ we have $s'_j = s_j$. We have that $\Phi(S')$ is exactly $\Phi(S)$ where one local machine moved from $\langle \rho, \ell\rangle$ to $\langle \rho, \ell_2\rangle$.

  Moreover, as $\Phi(S) = Created(S_G)$, there exists $1 \leq j \leq m$ such that $l_j = \langle \rho, \ell\rangle$ in S_G and the valuation of the global variable is $\Gamma \cup f(\Lambda)$. Following Rule 2b of section 3.2 and the semantics of global machines, S_G has one successor S'_G that is exactly S_G where one local machine moved from the local state $\langle\rho,\ell\rangle$ to $\langle\rho,\ell_2\rangle$. So, $\Phi(S') = \mathit{Created}(S'_G)$;

(c) There exists $1 \leq i \leq n$ such that $s_i = \langle[\ell]\texttt{choice}\{B_1{:}u_{1,1},\ldots,u_{1,m_1}{:=}v_{1,1},\ldots,v_{1,m_1};\ldots B_k{:}u_{k,1},\ldots,u_{k,m_k}{:=}v_{k,1},\ldots,v_{k,m_k};\}\cdot[\ell']I'\cdot S,\rho\rangle$ and there exists $1 \leq m \leq k$ such that $\Gamma\cup\rho \models B_m$. This last condition tells us that the guard is satisfied for the mth assignment which just have to deal with. For each possible assigment, we have the same reasoning than for simple assignment (Point (a)).

(d) There exists $1 \leq i \leq n$ such that $s_i = \langle[\ell]\texttt{while}\ (B)\ \texttt{do}\ \{[\ell_1]I_1\cdot S_1\}\cdot[\ell_2]I_2\cdot S_2,\rho\rangle$ and

- either $\Gamma\cup\rho \models B$, and S' is such that $s'_i = \langle[\ell_1]I_1\cdot S_1\cdot[\ell]\texttt{while}\ (B)\ \texttt{do}\ \{[\ell_1]I_1\cdot S_1\}\cdot[\ell_2]I_2\cdot S_2,\rho\rangle$; (ii) $n = n'$, $\Gamma' = \Gamma$, $\Lambda' = \Lambda$ and (iii) for all $1 \leq j \leq n$ $(j \neq i)$ we have $s'_j = s_j$. In that case, $\Phi(S')$ is exactly $\Phi(S)$ where one local machine moved from $\langle\rho,\ell\rangle$ to $\langle\rho,\ell_1\rangle$.

  Moreover, as $\Phi(S) = \mathit{Created}(S_G)$ by hypothesis, there exists $1 \leq j \leq m$ such that $l_j = \langle\rho,\ell\rangle$ in S_G and the valuation of the global variable is $\Gamma\cup f(\Lambda)$. Following Rule 2d of Section 3.2 and the semantics of global machines, S_G has one successor S'_G that is exactly S_G where one local machine moved from $\langle\rho,\ell\rangle$ to $\langle\rho,\ell_1\rangle$. So, $\Phi(S') = \mathit{Created}(S'_G)$.
- Otherwise $\Gamma\cup\rho \not\models B$, and S' is such that $s'_i = \langle[\ell_2]I_2\cdot S_2,\rho\rangle$; (ii) $n = n'$, $\Gamma' = \Gamma$, $\Lambda' = \Lambda$ and (iii) for all $1 \leq j \leq n$ $(j \neq i)$ we have $s'_j = s_j$. In that case, $\Phi(S')$ is exactly $\Phi(S)$ where one local machine moved from $\langle\rho,\ell\rangle$ to $\langle\rho,\ell_2\rangle$.

  Moreover, as $\Phi(S) = \mathit{Created}(S_G)$ by hypothesis, there exists $1 \leq j \leq m$ such that $l_j = \langle\rho,\ell\rangle$ in S_G and the valuation of the global variable is $\Gamma\cup f(\Lambda)$. Following Rule 2d of Section 3.2 and the semantics of global machines, S_G has one successor S'_G that is exactly S_G where one local machine moved from $\langle\rho,\ell\rangle$ to $\langle\rho,\ell_2\rangle$. So, $\Phi(S') = \mathit{Created}(S'_G)$.

(e) There exists $1 \leq i \leq n$ such that (i)$s_i = \langle[\ell]\texttt{skip};[\ell']I\cdot S,\rho\rangle$ and $s'_i = \langle[\ell']I\cdot S,\rho\rangle$ and; (ii) $n' = n$, $\Gamma' = \Gamma$ and $\Lambda' = \Lambda$; (iii) for all $1 \leq j \leq n$ $(j \neq i)$ we have $s'_j = s_j$. In that case, $\Phi(S')$ is exactly $\Phi(S)$ where one local machine moved from $\langle\rho,\ell\rangle$ to $\langle\rho,\ell'\rangle$.

   Moreover, as $\Phi(S) = \mathit{Created}(S_G)$ by hypothesis, there exists $1 \leq j \leq m$ such that $l_j = \langle\rho,\ell\rangle$ in S_G. Following Rule 2e of Section 3.2 and the semantics of global machines, S_G has one successor S'_G that is exactly S_G where one local machine moved from $\langle\rho,\ell\rangle$ to $\langle\rho,\ell'\rangle$. So, $\Phi(S') = \mathit{Created}(S'_G)$.

(f) There exists $1 \leq i \leq n$ such that (i)$s_i = \langle[\ell]\texttt{goto}(\ell');\cdot S,\rho\rangle$ and $s'_i = \langle[\ell']I\cdot S,\rho\rangle$; (ii) $n' = n$, $\Gamma' = \Gamma$ and $\Lambda' = \Lambda$; (iii) for all $1 \leq j \leq n$ $(j \neq i)$ we have $s'_j = s_j$. In that case, $\Phi(S')$ is exactly $\Phi(S)$ where one local machine moved from $\langle\rho,\ell\rangle$ to $\langle\rho,\ell'\rangle$.

   Moreover, as $\Phi(S) = \mathit{Created}(S_G)$ by hypothesis, there exists $1 \leq j \leq m$ such that $l_j = \langle\rho,\ell\rangle$ in S_G. Following Rule 2f of Section 3.2 and the semantics of global machines, S_G has one successor S'_G that is exactly S_G where one local machine moved from $\langle\rho,\ell\rangle$ to $\langle\rho,\ell'\rangle$. So, $\Phi(S') = \mathit{Created}(S'_G)$.

(g) There exists $1 \leq i \leq n$ such that (i)$s_i = \langle[\ell]\texttt{lock}(L);[\ell']I\cdot S,\rho\rangle$ and $s'_i = \langle[\ell']I\cdot S,\rho\rangle$; (ii) $n' = n$, $\Gamma' = \Gamma$ and $\Lambda' = \Lambda[L = \mathit{locked}]$; (iii) for all $1 \leq j \leq n$ $(j \neq i)$ we have $s'_j = s_j$.

In that case, $\Phi(S')$ is exactly $\Phi(S)$ where one local machine moved from $\langle \rho, \ell \rangle$ to $\langle \rho, \ell' \rangle$ and the global variable L (corresponding to the lock) is set to $true$.

Moreover, as $\Phi(S) = Created(S_G)$ by hypothesis, there exists $1 \leq j \leq m$ such that $l_j = \langle \rho, \ell \rangle$ in S_G and the valuation of the global variable is $\Gamma \cup f(\Lambda)$. Following Rule 2g of Section 3.2 and the semantics of global machines, S_G has one successor S'_G that is exactly S_G where one local machine moved from $\langle \rho, \ell \rangle$ to $\langle \rho, \ell' \rangle$ and the valuation becomes $\Gamma \cup f(\Lambda)[L = true]$. So, $\Phi(S') = Created(S'_G)$.

(h) There exists $1 \leq i \leq n$ such that (i)$s_i = \langle [\ell]\mathtt{unlock}(L); [\ell']I \cdot S, \rho \rangle$ and $s'_i = \langle [\ell']I \cdot S, \rho \rangle$; (ii) $n' = n$, $\Gamma' = \Gamma$ and $\Lambda' = \Lambda[L = unlocked]$; (iii) for all $1 \leq j \leq n$ $(j \neq i)$ we have $s'_j = s_j$. In that case, $\Phi(S')$ is exactly $\Phi(S)$ where one local machine moved from $\langle \rho, \ell \rangle$ to $\langle \rho, \ell' \rangle$ and the global variable L (corresponding to the lock) is set to $false$.

Moreover, as $\Phi(S) = Created(S_G)$ for hypothesis, there exists $1 \leq j \leq m$ such that $l_j = \langle \rho, \ell \rangle$ in S_G. Following Rule 2h of Section 3.2 and the semantics of global machines, S_G has one successor S'_G that is exactly S_G where one local machine moved from $\langle \rho, \ell \rangle$ to $\langle \rho, \ell' \rangle$ and the valuation becomes $\Gamma \cup f(\Lambda)[L = false]$ So, $\Phi(S') = Created(S'_G)$.

(i) There exists $1 \leq i \leq n$ such that: (i) $s_i = \langle [\ell]\mathtt{sleep}(M, L) \ ; \cdot \ [\ell']I' \cdot S'', \rho \rangle$ and $s'_i = \langle [\mathbf{wait}_\ell] wait(M, L) \cdot [\ell']I' \cdot S'', \rho \rangle$; (ii) $n' = n$, $\Gamma' = \Gamma$, $\Lambda' = \Lambda[L = unlocked]$; (iii) for all $1 \leq j \leq n$ $(j \neq i)$ we have $s'_j = s_j$. In that case, $\Phi(S')$ is exactly $\Phi(S)$ where one local machine moved from $\langle \rho, \ell \rangle$ to $\langle \rho, \mathbf{wait}_\ell \rangle$ and the global variable L is set to $false$.

Moreover, as $\Phi(S) = Created(S_G)$, there exists $1 \leq j \leq m$ such that $l_j = \langle \rho, \ell \rangle$ in S_G. Following Rule 2i of Section 3.2 and the semantics of global machines, S_G has one successor S'_G that is exactly S_G where one local machine moved from $\langle \rho, \ell \rangle$ to $\langle \rho, \mathbf{wait}_\ell \rangle$ and the valuation becomes $\Gamma \cup f(\Lambda)[L = false]$ So, $\Phi(S') = Created(S'_G)$.

(j) In the case of the `wakeup`, there are two cases to take into account, whether there are sleeping threads waiting to be woken up or not.

(a) In the former case, there exist $1 \leq i \neq j \leq n$ such that (i):

$$\begin{cases} s_i = \langle [\mathbf{wait}_\ell] wait(M, L) \cdot [\ell']I' \cdot S'', \rho_i \rangle \\ s'_i = \langle [\mathbf{lock}_\ell]\mathtt{lock}(L) \ ; \cdot \ [\ell']I' \cdot S'', \rho_i \rangle \\ s_j = \langle [\ell_j]\mathtt{wakeup}(M) \ ; \cdot \ [\ell'_j]I'_j \cdot S''_j, \rho_j \rangle \\ s'_j = \langle [\ell'_j]I'_j \cdot S''_j, \rho_j \rangle \end{cases}$$

and (ii) $n' = n$, $\Gamma' = \Gamma$, $\Lambda' = \Lambda$, and (iii) for all $1 \leq m \leq n : (m \notin \{i, j\})$, we have $s'_m = s_m$. In that case, $\Phi(S')$ is exactly $\Phi(S)$ where one local machine moved from $\langle \rho_i, \mathbf{wait}_\ell \rangle$ to $\langle \rho_i, \mathbf{lock}_\ell \rangle$ and one local machine moved from $\langle \rho_j, \ell_j \rangle$ to $\langle \rho_j, \ell'_j \rangle$.
Moreover, as $\Phi(S) = Created(S_G)$ by hypothesis, there exists $1 \leq i' \neq j' \leq m$ such that $l_{i'} = \langle \rho_i, \mathbf{wait}_\ell \rangle$ and $l_{j'} = \langle \rho_j, \ell_j \rangle$ in S_G. Following Rule 2j and Rule 2i of Section 3.2 and the semantics of global machines, S_G has one successor S'_G that is exactly S_G where one local machine moved from $\langle \rho_i, \mathbf{wait}_\ell \rangle$ to $\langle \rho_i, \mathbf{lock}_\ell \rangle$ and one local machine moved from $\langle \rho_j, \ell_j \rangle$ to $\langle \rho_j, \ell'_j \rangle$. So, $\Phi(S') = Created(S'_G)$.

(b) In the latter case, there exists $1 \leq i \leq n$ such that (i) $s_i = \langle [\ell]\mathtt{wakeup}(M) \ ; \cdot \ [\ell']I' \cdot S'', \rho \rangle$ and $s'_i = \langle [\ell']I' \cdot S'', \rho \rangle$; (ii) $n' = n$, $\Gamma' = \Gamma$ and $\Lambda' = \Lambda$; (iii) for all $1 \leq j \leq n \ : \ (j \neq i)$ the program remaining to be executed in s_j does not begin with the instruction $wait(M, L)$ for some lock L and we have $s'_j = s_j$. In that case, $\Phi(S')$ is exactly $\Phi(S)$ where one local machine moved from $\langle \rho, \ell \rangle$ to $\langle \rho, \ell' \rangle$.

Moreover, as $\Phi(S) = \mathit{Created}(S_G)$ by hypothesis, there exists $1 \leq i' \leq m$ such that $l_{i'} = \langle \rho, \ell \rangle$ in S_G and there is no $j' \neq i'$ such that $1 \leq j' \leq m$ and $l_{j'} = \langle \mathbf{wait}_{\ell''}, \rho' \rangle$ for some valuation ρ' and where the label ℓ'' labels the command $\texttt{sleep}(M, L)$ for some lock L in the CBP.

Following Rule 2j and Rule 2i (which is the only rule that adds transitions with signal reception in local machines) of Section 3.2 and the semantics of global machines, S_G has one successor S'_G that is exactly S_G where one local machine moved from $\langle \rho, \ell \rangle$ to $\langle \rho, \ell' \rangle$. So, $\Phi(S') = \mathit{Created}(S'_G)$.

(k) In the case of the `wakeupall`, we reorder and divide the set $\{s_1, \ldots s_n\}$ in three subsets $\Sigma_1 = \{s_1, \ldots, s_k\}$, $\Sigma_2 = \{s_{k+1}, \ldots, s_{n-1}\}$ and $\Sigma_3 = \{s_n\}$, such that: (i), for every $1 \leq i \leq k$: $s_i = \langle [\mathbf{wait}_{\ell_i}] wait(M, L_i) \cdot [I'_i].S''_i, \rho_i \rangle$ for some lock L_i; (ii), for every $k < j < n$: the program S_j remaining to be executed in s_j does *not* begin with the command $wait(M, L)$ for some lock L and (iii) s_n is the state of the local machine doing the `wakeupall`. More precisely, $s_n = \langle [\ell_n]\texttt{wakeupall}(M) \ ; \cdot \ [\ell'_n] I'_n \cdot S''_n, \rho_n \rangle$. Notice that Σ_1 and Σ_2 can be empty. S' is such that (i) for all $1 \leq i \leq k : s'_i = \langle [I'_i].S''_i, \rho_i \rangle$; for every $k < j < n : s'_j = s_j$ and $s'_n = \langle [\ell'_n] I'_n \cdot S''_n, \rho_n \rangle$; (ii) $n' = n$, $\Gamma' = \Gamma$, $\Lambda' = \Lambda$. In that case, $\Phi(S')$ is exactly $\Phi(S)$ where one local machine moved from $\langle \rho_n, \ell_n \rangle$ to $\langle \rho_n, \ell'_n \rangle$ and k local machines moved such that the *ith* local machine moved from the local state $\langle \rho_i, \mathbf{wait}_{\ell_i} \rangle$ to $\langle \rho_i, \mathbf{lock}_{\ell_i} \rangle$.

Moreover, as $\Phi(S) = \mathit{Created}(S_G)$ by hypothesis, there exist $1 \leq j \leq m$ such that $l_j = \langle \rho_n, \ell_n \rangle$ in S_G (where ℓ_n labels a $\texttt{wakeupall}(M)$ command in B) and k local machines such that the *ith* local machine is in a local state of the form $\langle \rho_i, \mathbf{wait}_{\ell_i} \rangle$ (where the label ℓ_i labels the command $\texttt{sleep}(M, L)$ for some lock L in the CBP).

Following Rule 2k and Rule 2i (which is the only rule that adds transitions with broadcast reception in local machines) of Section 3.2 and the semantics of global machines, S_G has one successor S'_G that is exactly S_G where one local machine moved from $\langle \rho_n, \ell_n \rangle$ to $\langle \rho_n, \ell'_n \rangle$ and k local machines moved such that the *ith* local machine moved from the local state $\langle \rho_i, \mathbf{wait}_{\ell_i} \rangle$ to $\langle \rho_i, \mathbf{lock}_{\ell_i} \rangle$. So, $\Phi(S') = \mathit{Created}(S'_G)$.

(l) There exists $1 \leq i \leq n$ such that (i)$s_i = \langle [\ell]\texttt{start}(T, \pi_1, \ldots, \pi_m); [\ell'] I \cdot S, \rho \rangle$ and $s'_i = \langle [\ell'] I \cdot S, \rho \rangle$ and, (ii) $n' = n + 1$, $\Gamma' = \Gamma$ and $\Lambda' = \Lambda$, and (iii) for all $1 \leq j \leq n$ $(j \neq i)$ we have $s'_j = s_j$. Moreover, if T is the identifier of thread type $T_j = \langle \mathcal{V}_j, S_j \rangle$, then the state of the $n+1$th thread (the newly created one) is $s_{n+1} = \langle [\ell_{n+1}] I_{n+1} \cdot S'_{n+1}, \rho_{n+1} \rangle$, where:

- $S_j \equiv [\ell_{n+1}] I_{n+1} \cdot S'_{n+1}$;
- ρ_{n+1} is such that: $\forall 1 \leq l \leq m : \rho_{n+1}(v_l) = \pi_l$.

In that case, $\Phi(S')$ is exactly $\Phi(S)$ where one local machine moved from $\langle \rho, \ell \rangle$ to $\langle \rho, \ell' \rangle$ and where one local machine in the local state $\langle \rho', \ell_{n+1} \rangle$ where $\rho'(v_l) = \pi_l$ for all $v_l \in \mathcal{V}_j$ is added.

Moreover, as $\Phi(S) = \mathit{Created}(S_G)$ by hypothesis, there exists in S_G one local machine in the local state $\langle \rho, \ell \rangle$ and one local machine in its local state $\langle \theta_{\mathcal{L}_j, \mathbf{begin}_{\mathcal{L}_j}} \rangle$ since by hypothesis $NT(S', T_j) \leq k_j$ where k_j is the number of local machines corresponding to the thread type T_j in G_B.

Thanks to the construction rules explained in section 3.2 (rule 2l and the construction of transitions from local states of the form $\langle \theta_{\mathcal{L}_j, \mathbf{begin}_{\mathcal{L}_j}} \rangle$) and the semantics of global machines, S_G has one successor S'_G that is exactly S_G where one local machine moved from $\langle \rho, \ell \rangle$ to $\langle \rho, \ell' \rangle$ and one local machine moved $\langle \theta_{\mathcal{L}_j, \mathbf{begin}_{\mathcal{L}_j}} \rangle$ to $\langle \rho', \ell_{n+1} \rangle$ where $\rho'(v_l) = \pi_l$ for all $v_l \in \mathcal{V}_j$. So, $\Phi(S') = \mathit{Created}(S'_G)$.

(m & n) There exists $1 \leq i \neq j \leq n$ such that:

$$(i) \begin{cases} s_i = \langle [\ell_i]\texttt{rendezvous}(M, u_1, \ldots, u_n) \ ; \cdot \ [\ell'_i] I'_i \cdot S''_i, \rho_i \rangle \\ s'_i = \langle [\ell'_i] I'_i \cdot S''_i, \rho_i \rangle \\ s_j = \langle [\ell_j]\texttt{accept}(M, v_1, \ldots, v_n) \ ; \cdot \ [\ell'_j] I'_j \cdot S''_j, \rho_j \rangle \\ s'_j = \langle [\ell'_j] I'_j \cdot S''_j, \rho_j[v_1 = u_1, \ldots v_n = u_n] \rangle \end{cases}$$

(ii) $n' = n$, $\Gamma' = \Gamma$, $\Lambda' = \Lambda$, and (iii) for all $1 \leq m \leq n : m \neq i$ and $m \neq j$, we have $s'_m = s_m$.

In that case, $\Phi(S')$ is exactly $\Phi(S)$ where one local machine moved from $\langle \rho_i, \ell_i \rangle$ to $\langle \rho_i, \ell'_i \rangle$ and one local machine moved from $\langle \rho_j, \ell_j \rangle$ to $\langle \rho'_j, \ell'_j \rangle$ where $\rho'_j = \rho_j[v_1 = u_1 \ldots v_n = u_n]$.

Moreover, as $\Phi(S) = \mathit{Created}(S_G)$ by hypothesis, there exists $1 \leq i' \neq j' \leq m$ such that $l_{i'} = \langle \rho_i, \ell_i \rangle$ and $l_{j'} = \langle \rho_j, \ell_j \rangle$ in S_G. Following Rule 2m and Rule 2n of Section 3.2 and the semantics of global machines, S_G has one successor S'_G that is exactly S_G where one local machine moved from $\langle \rho_i, \ell_i \rangle$ to $\langle \rho_i, \ell'_i \rangle$ and one local machine moved from $\langle \rho_j, \ell_j \rangle$ to $\langle \rho'_j, \ell'_j \rangle$ where $\rho'_j = \rho_j[v_1 = u_1 \ldots v_n = u_n]$. So, $\Phi(S) = \mathit{created}(S'_G)$.

Since there exists a successor S'_G of S_G such that $\Phi(S') = \mathit{Created}(S'_G)$, we conclude that there exists a run $R' = R \cdot S_G \cdot S'_G$ such that $\Phi(e) = \mathit{Created}(R')$. $\square$

.4 Code of Java classes and corresponding CBP code.

```
Class SimpleTransBankAccount {

  private long balance = 0;
  private long workingBalance;
  Thread lockref = null;

  public synchronized long balance(Thread t) {
    if (lockref != t)
      join(t);
    return workingBalance;
  }
  public synchronized void deposit(Thread t,long amount) {
    if (lockref != t)
      join(t);
    workingBalance += amount;
  }
  public synchronized void withdraw(Thread t,long amount) {
    deposit(t,-amount);
  }
  public synchronized void join(Thread t) {
    while (lockref != NULL) {
      wait();
    }
    lockref = t;
    workingBalance = balance;
  }
  public synchronized boolean canCommit(Thread t) {
    return (lockref == t);
  }
  public synchronized void abort(Thread t) {
    if (lockref == t) {
      lockref = null;
      notifyAll();
    }
  }
  public synchronized void commit(Thread t) {
    if (t == lockref) {
      balance = workingBalance;
      lockref = null;
      notifyAll();
    }
  }
}
```

Figure 1: A class for transactions of account databases.

```
class Tansactionthread extends Threads {
  private Vector V;

  public Tansactionthread(Vector V) {
    this.V  = V;
  }
  public void run() {
    Random R = new Random();
    int command = 0;
    int obj = 0;
    int balance;
    boolean bool
    while ((command != 5) && (command != 6)) {
      command = R.nextInt(6);
      obj = R.nextInt(V.capacity());
      if (command == 0) {
        balance = ((SimpleTransBankAccount)V.get(obj)).balance(this);
      } else if (command == 1) {
        ((SimpleTransBankAccount)V.get(obj)).deposit(this,5);
      } else if (command == 2) {
        ((SimpleTransBankAccount)V.get(obj)).withdraw(this,5);
      } else if (command == 3) {
        ((SimpleTransBankAccount)V.get(obj)).join(this);
      } else if (command == 4) {
        bool = ((SimpleTransBankAccount)V.get(obj)).cancommit(this);
      } else if (command == 5) {
        for(obj = 0;obj < V.capacity();obj++) {
          ((SimpleTransBankAccount)V.get(obj)).abort(this);}
      } else if (command == 6) {
        for(obj = 0;obj < V.capacity();obj++) {
          ((SimpleTransBankAccount)V.get(obj)).commit(this);
      }
    }
  }
}
```

Figure 2: JAVA thread class that manipulates SimpleTransBankAccount objects.

```
TransactionthreadA {var :;
  while(*) {
    if (*) {                    //call to balance()
      lock(lock0);
      if (!lockref_eq_A) {
        while(!lockref_eq_null) {
          sleep(msg0,lock0);
        }
        lockref_eq_A, lockref_eq_null := true, false;
      }
      unlock(lock0);
    } else {

       ...                      //call to deposit() and withdraw()

          } else {
            if (*) {
              lock(lock0);
              while(!lockref_eq_null) {
                sleep(msg0,lock0);
              }
              lockref_eq_A, lockref_eq_null := true, false;
              unlock(lock0);
            } else {
              if (*) {          // call to cancommit()
                lock(lock0);
                unlock(lock0);
              }
              else {
                if (*) {        //call to abort
                  lock(lock0);
                  if (lockref_eq_A) {
                    lockref_eq_A, lockref_eq_numm := false, true;
                    wakeupall(msg0);
                  } else { skip; }
                  unlock(lock0);
                } else {

                 ...           //call to commit
} } } } } } } } }
```

Figure 3: CBP thread corresponding to the JAVA thread instance A of the class Tansaction-thread.

```
TransactionthreadO {var :;
  while(*) {
    if (*) {                            //call to balance()
      lock(lockO);
      if (lockref_eq_A or lockref_eq_null) {
        while(lockref_eq_null) {
          sleep(msgO,lockO);
        }
        lockref_eq_A,lockref_eq_null := false,false;
      } else {
        if(*) {
          while(lockref_eq_null) {
            sleep(msgO,lockO);}
          lockref_eq_A,lockref_eq_null := false,false;
        } else {skip;}
      }
      unlock(lockO);
    }
    else {
      ...                              //call to deposit() and withdraw()
          } else {
            if (*) {                   //call to join
              lock(lockO);
              while(lockref_eq_null) {
                sleep(msgO,lockO);}
              lockref_eq_A,lockref_eq_null := false,false;
              unlock(lockO);
            } else {
              if (*) {                 //call to cancommit
                lock(lockO);
                unlock(lockO);
              } else {                 //call to abort
                if (*) {
                  lock(lockO);
                  if (!lockref_eq_A and !lockref_eq_null) {
                    if (*) {
                      lockref_eq_null, lockref_eq_null := true, false;
                      wakeupall(msgO);
                    } else {skip; }
                  } else { skip; }
                  unlock(lockO);
                } else {
                  ...                  //call to commit
} } } } } } } } }
```

Figure 4: CBP thread corresponding to the JAVA thread instances of the class Tansactionthread.

```
public class Boundedbuffer {
  private Object[] array;
  private Exchanger putter;
  private Exchanger taker;

  public Boundedbuffer(int Capacity) {
    int c;
    if (capacity <= 0)
      c = 1; //if an illegal value is given, then
             //the buffer will have the default size 1
    else
      c = capacity;
    array = new Object[c];
    putter = new Exchanger[c];
    taker =  new Exchanger[c];
  }

  public void put(Object x) {
    if (x != null)
      putter.exchange(x);
  }

  public Object take() {
    return taker.exchange(null);
  }

  void removeSlotNotification(Exchanger h) {
    if (h == putter)
      taker.addedSlotNotification();
    else
      putter.addSlotNotification();
  }
```

Figure 5: The JAVA Boundedbuffer class.

```
  protected class Exchanger {
    private int ptr = 0;
    private int slot;
    private int waiting = 0;

    public Exchanger(int n) {
      slot = n;
    }

    synchronized void addedSlotNotification() {
      ++slots;
      if (waiting > 0)
        notify();
    }

    Object exchange(Object x) {
      Object old = null;

      synchronized(this) {
        while(slots <= 0) {
          ++ waiting;
  wait();
          --waiting;
        }
        --slot;
        old = array[ptr];
        array[ptr] = x;
        ptr = (ptr + 1) % array.lenght;
      }
      removeSlotNotification(this);
      return old;
    }
  }
}
```

Figure 6: The JAVA Exchanger class (defined inside the Boundedbuffer class).

```
public class Putthread extends Threads {

 Boundedbuffer buffer;

  public Putthread(Boundedbuffer buffer) {
    this.buffer = buffer;
  }

  public void run() {
    Object O;

    while(true) {
      O = new Object();
      buffer.put(O);
    }
  }
}

public class Takethread extends Threads {

  Boundedbuffer buffer;

  public Takethread(Boundedbuffer buffer) {
    this.buffer = buffer;
  }

  public void run() {
    Obeject O;

    while(true) {
      O = buffer.take();
    }
  }
}
```

Figure 7: JAVA thread classes that manipulates a Boundedfuffer object.

```
Threadtaker { vars: ;
  while (true) {
    lock(lockputter);
    while (!Pslot1 and !Pslot2 and !Pslot3) {
      sleep(msgputter,lockputter);
    }
    choice {
      Pslot0 : Pslot0, Pslot1, Pslot2, Pslot3:=false,false,false,false;
      Pslot1 : Pslot0, Pslot1, Pslot2, Pslot3:=true,false,false,false;
      Pslot2 : Pslot0, Pslot1, Pslot2, Pslot3:=false,true,false,false;
      Pslot3 : Pslot0, Pslot1, Pslot2, Pslot3:=false,false,true,false;
      !Pslot0 and !Pslot1 and !Pslot2 and !Pslot3:=false,false,false,true;
      !Pslot0 and !Pslot1 and !Pslot2 and !Pslot3:=false,false,false,true;}
    choice {
      Pptr0 : Pptr0, Pptr1, Pptr2:=false,true,false;
      Pptr1 : Pptr0, Pptr1, Pptr2:=false,false,true;
      Pptr2 : Pptr0, Pptr1, Pptr2:=true,false,false;
      !Pptr0 and !Pptr1 and !Pptr2:=true,false,false; //if ptr = -1
      !Pptr0 and !Pptr1 and !Pptr2:=false,false,false;} //the other cases
    unlock(lockputter);
    if (true) {
      lock(locktaker);
      choice {
        Tslot0 : Tslot0, Tslot1, Tslot2, Tslot3:=false,true,false,false;
        Tslot1 : Tslot0, Tslot1, Tslot2, Tslot3:=false,false,true,false;
        Tslot2 : Tslot0, Tslot1, Tslot2, Tslot3:=false,false,false,true;
        Tslot3 : Tslot0, Tslot1, Tslot2, Tslot3:=false,false,false,false;
        !Tslot0 and !Tslot1 and !Tslot2 and !Tslot3:=true,false,false,false;
        !Tslot0 and !Tslot1 and !Tslot2 and !Tslot3:=false,false,false,false;}
       wakerup(msgtaker);
       unlock(locktaker);
    } else {
      lock(lockputter);
      choice {
        Pslot0 : Pslot0, Pslot1, Pslot2, Pslot3:=false,true,false,false;
        Pslot1 : Pslot0, Pslot1, Pslot2, Pslot3:=false,false,true,false;
        Pslot2 : Pslot0, Pslot1, Pslot2, Pslot3:=false,false,false,true;
        Pslot3 : Pslot0, Pslot1, Pslot2, Pslot3:=false,false,false,false;
        !Pslot0 and !Pslot1 and !Pslot2 and !Pslot3:=true,false,false,false;
        !Pslot0 and !Pslot1 and !Pslot2 and !Pslot3:=false,false,false,false;}
       wakeup(msgputter);
       unlock(lockputter);
    }
  }
```

Figure 8: The CBP thread corresponding the the JAVA Threadputter class.

```
Threadtaker { vars: ;
  while (true) {
    lock(locktaker);
    while (!Tslot1 and !Tslot2 and !Tslot3) {
      sleep(msgtaker,locktaker);
    }
    choice {
      Tslot0 : Tslot0, Tslot1, Tslot2, Tslot3:=false,false,false,false;
      Tslot1 : Tslot0, Tslot1, Tslot2, Tslot3:=true,false,false,false;
      Tslot2 : Tslot0, Tslot1, Tslot2, Tslot3:=false,true,false,false;
      Tslot3 : Tslot0, Tslot1, Tslot2, Tslot3:=false,false,true,false;
      !Tslot0 and !Tslot1 and !Tslot2 and !Tslot3:=false,false,false,true;
      !Tslot0 and !Tslot1 and !Tslot2 and !Tslot3:=false,false,false,true;}
    choice {
      Tptr0 : Tptr0, Tptr1, Tptr2:=false,true,false;
      Tptr1 : Tptr0, Tptr1, Tptr2:=false,false,true;
      Tptr2 : Tptr0, Tptr1, Tptr2:=true,false,false;
      !Tptr0 and !Tptr1 and !Tptr2:=true,false,false; //if ptr = -1
      !Tptr0 and !Tptr1 and !Tptr2:=false,false,false;} //the other cases
    unlock(locktaker);
    if (false) {
      lock(locktaker);
      choice {
        Tslot0 : Tslot0, Tslot1, Tslot2, Tslot3:=false,true,false,false;
        Tslot1 : Tslot0, Tslot1, Tslot2, Tslot3:=false,false,true,false;
        Tslot2 : Tslot0, Tslot1, Tslot2, Tslot3:=false,false,false,true;
        Tslot3 : Tslot0, Tslot1, Tslot2, Tslot3:=false,false,false,false;
        !Tslot0 and !Tslot1 and !Tslot2 and !Tslot3:=true,false,false,false;
        !Tslot0 and !Tslot1 and !Tslot2 and !Tslot3:=false,false,false,false;}
       wakeup(msgtaker);
       unlock(locktaker);
    } else {
      lock(lockputter);
      choice {
        Pslot0 : Pslot0, Pslot1, Pslot2, Pslot3:=false,true,false,false;
        Pslot1 : Pslot0, Pslot1, Pslot2, Pslot3:=false,false,true,false;
        Pslot2 : Pslot0, Pslot1, Pslot2, Pslot3:=false,false,false,true;
        Pslot3 : Pslot0, Pslot1, Pslot2, Pslot3:=false,false,false,false;
        !Pslot0 and !Pslot1 and !Pslot2 and !Pslot3:=true,false,false,false;
        !Pslot0 and !Pslot1 and !Pslot2 and !Pslot3:=false,false,false,false;}
       wakeup(msgputter);
       unlock(lockputter);
    }
  }
```

Figure 9: The CBP thread corresponding the the JAVA Threadtaker class.