Université Libre de Bruxelles

Faculté des sciences

Département d'informatique

$_{\mathrm{d}}$SL,
a Language and Environment for the Design of Distributed Industrial Controllers

Bram De Wachter

November, 2005

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

Abstract

We present $_{\mathrm{d}}$SL (distributed Supervision Language), a language and environment dedicated to the specification of industrial controllers. $_{\mathrm{d}}$SL extends an existing language used in the industrial world to allow transparent code distribution. We study other existing solutions, and argue for the need of $_{\mathrm{d}}$SL. Next, we define $_{\mathrm{d}}$SL syntactically and semantically, and prove some remarkable properties on its semantics. The automatic distribution introduces problems that are hard to solve for the $_{\mathrm{d}}$SL compiler/distributer, for which we give some heuristics. Since we give a precise semantics to the $_{\mathrm{d}}$SL language, formal methods can be applied to assure that controllers are correct. We show how to use explicit state model checking to perform this verification. Throughout the presentation, we introduce a set of examples showing the useability of $_{\mathrm{d}}$SL and its concepts.

Keywords: Distributed systems, language design, compilation, reactive systems, formal methods.

Contents

Acknowledgments

I would like to thank my promoter Thierry Massart for offering me the opportunity and grants that allowed me to work at the *Université Libre de Bruxelles*. Without this opportunity, this work would not have seen the light of day.

Thanks to all members of the jury for their time and precious feedback. The pertinence of their remarks greatly improved this thesis.

I also would like to thank the research and development department of Macq Electronique. Geert, Bernard and Jean-François thank you for your involvement in this work. Your valuable expertise and the many hours of discussion had a major impact on the practical side of this work.

Many grateful thanks go to my fellow researchers, and more particularly Cédric Meuter and Alexandre Genon, for those fruitful moments of cooperation and discussions that shaped this work.

I could not have managed without the support and the hearth warming encouragements of all co-workers of the Verification Group. I hope that those touching moments of fun and fraternity will not fade in memory. I can not imagine (re)reading this thesis and not thinking of each single one of you.

Special thanks go to Laurent Van Begin and Clément Daniel. I am unable to express how much their presence meant to me. They know.

En laatst, maar niet in het minst, wil ik een oprechte en welgemeende dank aan mijn familie richten. Jullie waren er steeds opnieuw met raad, liefde en daad. Vava en Moemoe, Vake, Moeke, Nele, Griet en Wouter, ... zonder jullie zou het voor mij niet mogelijk zijn geweest. Dank je.

Bram De Wachter

Chapter 1

Introduction

Context

This work is situated in the world of industrial process control, which aims at managing real world environments such as traffic lights, domotica, road signaling, power plant regulations, assembly belts, ... The industrial controllers used to manage these environments observe, decide and react. A controller observes the environment through a set of captors, which transform the physical state of the environment into data ready to be handled by the controller. The intelligent part of the controller makes decisions based upon these measures, hence dictating the controller's behavior. A controller reacts to the environment, using the outcome of these decisions, through actuators which are hardware devices capable of forcing some physical state of the environment. For a system controlling traffic lights, sensors indicate the arrival of cars, while actuators consist in green, red and yellow light bulbs.

Industrial process control goes hand in hand with distributed systems. This is due to the physically distributed nature of the environment, where various sensors and actuators may span large distances. Several controllers, each connected to a set of nearby sensors and actuators, must therefore cooperate to control the environment. The development of such a network of controllers is a complicated task, even for experienced programmers. The burden of combining the physical complexity of the process to control, the communication schemes of the distributed parts, the need to provide simple and fast control and the extreme reliability and robustness requirements make the development of such systems hard.

The need for techniques alleviating the additional task inflicted by the physical distribution is therefore significant.

Goal

To simplify the work of the distributed system's designer, we introduce a novel programming language with transparent code distribution, hence the programmer is no longer burdened with the distributed aspects of the system, and can therefore concentrate on its functional aspects.

In addition to removing the physical distribution from the designer's task, we aim at providing the programmer with a way of establishing functional correctness of the designed system. The nature of the systems considered in this work requires absolute confidence in their behavioral correctness. Often these systems are used to control high risk environments where an error in the controller may have tremendous consequences. The financial and human impact of an error in an assembly line controller or the catastrophe due to a malfunctioning controller in a power plant makes the correctness of these systems indispensable.

A third important goal is that our language must take into consideration the languages that are actually used in the industry. From the theoretical side, the above mentioned problems have long been studied before, and many solutions have been proposed. Our language is not intended to be yet another automatically distributed design language for reactive systems. What is unique in this study, is that the design of our language is based upon an existing industrial language, which in its turn is based upon an industrial standard.

Constraints

To reach the above mentioned goals, several constraints must be taken into account. These can be summarized in the following three concerns.

First, there is an efficiency concern since programs designed with our new language must be able to execute on industrial controllers which have very limited resources. The limited amount of available processing power and memory on the controllers requires a design that excels in its simplicity.

Next, a major concern for industrial control systems is the ability to easily monitor the system. Otherwise stated, the system should be designed in such a way that one knows, at any time, how the controllers are managing the environment.

Finally, we are faced with a robustness concern. More particularly, each controller should be, in some sense, independent of the other controllers. The breakdown of one controller in the system should have only limited effect on the other controllers.

dSL

dSL is a domain specific language designed to program industrial controllers providing transparent distribution. This transparent distribution offers the programmer a view of the environment were all actuators and sensors are connected to a single controller. The task of the programmer is therefore reduced to writing a single program, in contrast to writing a set of cooperating programs (one for each controller in the system). The specification of the behavior of a distributed control system in dSL is basically a two step process :

1. The functional design of the system is specified in a single program written in dSL.

2. The programmer specifies the physical location of the actuators and sensors, possibly connected to different controllers. This is done in a localization table, which is not part of the program.

The dSL program and the localization table are then fed into the dSL compiler-distributer which produces a set of cooperating programs performing the control specified by the programmer.

Since the keyword in the world of control system is robustness, the accent in the design of dSL has been put on simplicity. We shall see, e.g. that dynamic features have been avoided as much as possible to allow an easy monitoring during the execution.

Additionally, dSL has a formal semantics which allows us to give a mathematical description of the behavior of a program written in dSL. This mathematical model can then be used by verification tools which are able to formally prove the correctness of the system with respect to a certain property. For a traffic light controller, these tools can for example insure that the controller is unable to turn on the green lights for crossing directions of an intersection at the same time.

Advantages

The practical advantages of using transparent code distribution mechanisms are obvious. First, maintainability is increased since only a single program is used to specify the behavior of the controller. Next, flexibility is increased, since the reconfiguration of actuators and sensors does only require the programmer to change the localization table, as opposed to changing the program. Finally, the simplicity in the design and our formal approach

allows to achieve a level of confidence in the correctness of the controller's behavior that could never be achieved without it.

Aspects covered in this study

This thesis covers a wide range of different aspects in computer science and results can surely be used outside the world of process control.

The first element is related to software engineering, with the design of $_{\text{d}}$SL. Some language aspects in $_{\text{d}}$SL could be improved from a theoretical point of view. However, bear in mind that this work starts from what is used in practice (a programming language in contrast to a design language), and proposes a solution with practical applicability in mind.

Secondly, we enter the domain of formal methods, with the introduction of $_{\text{d}}$SL's semantics. These formal semantics are also used to establish functional correctness of controllers programmed in $_{\text{d}}$SL, using a technique called model-checking.

A third element explored in this thesis comes from graph theory and algorithmics, when we introduce and study the algorithmic difficulties introduced by the transparent distribution of $_{\text{d}}$SL. New results for a well known problem are presented, yielding new insights which open new ways to tackle this problem.

We also study elements belonging to the distributed systems area of research. The execution environment of our language uses a technique called thread migration which, in this work, is adapted for our specific needs.

Finally, the implementation of the $_{\text{d}}$SL compiler-distributer combines all these matters together, using additional elements from compilation theory.

Plan of the thesis

In chapter 2, we discuss related work and show to what extent this work has an impact on the design of $_{\text{d}}$SL.

Next, in chapter 3, we present $_{\text{d}}$SL. We first introduce $_{\text{d}}$SL's ancestor SL (Supervision Language), a programming language used in the industry to program industrial controllers. We show how some syntactical additions to this language allow the transparent code distribution of $_{\text{d}}$SL. Next, we introduce $_{\text{d}}$SL's semantics, formalized as a labeled transition system. We show how the physical distribution of the environment influences this semantics, and introduce a relation between the different behaviors of a given $_{\text{d}}$SL program with respect to its physical distribution. We end this chapter

by introducing a set of examples. The presentation of $_{\mathrm{d}}$SL has also been presented in [DeWMM03a]. The semantics and its relation with the physical distribution are presented in [DeWGMM05].

In chapter 4, we study the algorithmic difficulties introduced by the transparent distribution of $_{\mathrm{d}}$SL. We show that these additional problems are difficult to solve by relating them to NP-Complete problems defined on graphs. Since these problems can not be solved in a reasonable amount of time, we suggest efficient heuristics. The work presented in this chapter was published in [DeWGM05].

Chapter 5 shows how the theoretical study of the $_{\mathrm{d}}$SL language is used in practice. We introduce some implementation aspects of the $_{\mathrm{d}}$SL compiler-distributer. This compiler-distributer transforms a $_{\mathrm{d}}$SL code into a set of cooperating executable codes. The $_{\mathrm{d}}$SL compiler-distributer presented in this chapter has been taken over by Macq Electronique who integrated it in their industrial toolsuite OBViews.

We show how the correctness of a $_{\mathrm{d}}$SL program can be automatically proved by using verification tools in chapter 6. We study how an explicit state model checker can be used to establish functional correctness with respect to a certain property. We therefore show how a $_{\mathrm{d}}$SL program can be transformed into a formal specification accepted by this model checker. We illustrate this approach with results in the verification of the examples given in chapter 3. Some of the verification results presented here are also published in [DeWMM03b].

Chapter 7 presents future works. This includes a discussion on some possible extensions of $_{\mathrm{d}}$SL, including a real-time semantics. We also re-explore the work presented in chapter 4. We end this chapter with a brief presentation of two techniques combining formal verification and testing.

Finally, Chapter 8 concludes this thesis.

Chapter 2

Motivation and other approaches

The problem of specifying reactive systems with transparent distribution has extensively been discussed in the literature, and several formal solutions to the problem have been proposed. Unfortunately, in the industrial world only very little effort is made in adopting these solutions. Either the proposed solutions are too abstract, or they have inherent technical issues that make them unusable in practice. To specify such systems, the industry therefore uses proprietary programming languages, which are generally based upon industrial standards.

In order to overcome these problems, we chose to marry both the formal world and the practical world. We therefore use an existing language used in the industry as a basis for our new language, extend its syntax and formalize its semantics using the observations made on the solutions found in the literature.

In this chapter, we give a critical overview of these existing formal solutions. The observations made in this chapter have a clear impact on the design of $_{\mathrm{d}}$SL's syntax and semantics which are presented, together with the industrial language used as the basis for $_{\mathrm{d}}$SL, in the next chapter. We comment in more detail on process algebra (section 2.1.1), higher level frameworks such as Unity (section 2.1.2), and synchronous languages (section 2.1.3). The choice of the distributed execution environment used by $_{\mathrm{d}}$SL is motivated by a discussion on distributed shared memory systems (section 2.2.1) and thread migration (section 2.2.2).

2.1 Solutions with transparent distribution

2.1.1 Process algebra

In the world of process algebras, the problem of automated distribution is defined as a correctness preserving transformation of a *centralized* specification into a semantically equivalent distributed one. More specifically, a centralized specification B, which has a certain set of actions A, is transformed into a specification containing two synchronizing processes B_1 and B_2 in such a way that the processes B_1 and B_2 only perform the actions in A_1 and A_2, (with $A = A_1 \cup A_2 \wedge A_1 \cap A_2 = \emptyset$). The bi-partition of A in A_1 resp. A_2 can for example be based on the geographical distribution of certain functionalities (actions) at different locations. The result of the composition of B_1 and B_2 must be equivalent, with respect to some equivalence relation, to the centralized specification B (e.g. for bisimulation equivalence [Mil89], see [Mas92, BL95], for observational equivalence [Mil89], see [BLB93]). Remark that more complicated subdivisions can be obtained by repeatedly applying this bi-partition.

We illustrate these concepts using the notations from [BLB93]. Let L be a set of action labels ($\{i, \delta\} \cap L = \emptyset$) and PN a set of process names. Let $g \in L \cup \{i\}, a \in L \cup \{i, \delta\}, G \subseteq L$, and $P \in PN$. Let H be a function $H : L \cup \{i, \delta\} \mapsto L \cup \{i, \delta\}$ with $H(i) = i$ and $H(\delta) = \delta$. The syntax of a process definition is $P := B$, where B is a behavior expression. A set of process definitions $\{P := B_P | P \in PN\}$ is called a process environment. A LOTOS [BB87] specification is a behavior expression in the context of a process environment. The syntax and operational semantics of Basic LOTOS behavior expressions is given in figure 2.1.

The problem can now be formally stated as follows : given a process behavior B with actions A, and a partition of A into A_1 and A_2, find processes B_1 and B_2 such that

$$(\textbf{hide } \{sync\} \textbf{ in } B_1 | [\{sync\}] | B_2) \approx B$$
$$\text{and Actions}(B_1) = A_1, \text{Actions}(B_2) = A_2$$

where *sync* is a new action ($sync \in L \setminus \text{Actions}(B)$) which enables B_1 and B_2 to synchronize, and $\approx$ expresses an equivalence relation.

Several solutions for this problem have been proposed. In order to shed a light on these solutions, we use the results from [BLB93], where $\approx$ stands for observational equivalence [Mil89]. In this paper, the transformation of B into B_1 and B_2 is defined using two mappings T_1 and T_2, such that $T_1(B) = B_1$, and $T_2(B) = B_2$. These mappings are recursively defined

inaction	**stop**	
termination	**exit**	**exit** $\xrightarrow{\delta}$ **stop**
action prefix	$g; B$	$g; B \xrightarrow{g} B$
choice	$B_1 [] B_2$	$B_1 \xrightarrow{a} B_1' \vdash B_1 [] B_2 \xrightarrow{a} B_1'$
		$B_2 \xrightarrow{a} B_2' \vdash B_1 [] B_2 \xrightarrow{a} B_2'$
enabling	$B_1 \gg B_2$	$B_1 \xrightarrow{a} B_1', a \neq \delta \vdash B_1 \gg B_2 \xrightarrow{a} B_1' \gg B_2$
		$B_1 \xrightarrow{\delta} B_1' \vdash B_1 \gg B_2 \xrightarrow{i} B_2$
disabling	$B_1 [> B2$	$B_1 \xrightarrow{a} B_1', a \neq \delta \vdash B_1 [> B_2 \xrightarrow{a} B_1' [> B_2$
		$B_1 \xrightarrow{\delta} B_1' \vdash B_1 [> B_2 \xrightarrow{\delta} B_1'$
		$B_2 \xrightarrow{a} B_2' \vdash B_1 [> B_2 \xrightarrow{a} B_2'$
hiding	**hide** G **in** B	$B \xrightarrow{a} B', a \in G \vdash$ **hide** G **in** $B' \xrightarrow{i}$ **hide** G **in** B'
		$B \xrightarrow{a} B', a \notin G \vdash$ **hide** G **in** $B' \xrightarrow{a}$ **hide** G **in** B'
renaming	$B[H]$	$B \xrightarrow{a} B' \vdash B[H] \xrightarrow{H(a)} B'[H]$
parallel	$B_1 \lvert[G]\rvert B_2$	$B_1 \xrightarrow{a} B_1', a \notin G \cup \{\delta\} \vdash B_1 \lvert[G]\rvert B_2 \xrightarrow{a} B_1' \lvert[G]\rvert B_2$
composition		$B_2 \xrightarrow{a} B_2', a \notin G \cup \{\delta\} \vdash B_1 \lvert[G]\rvert B_2 \xrightarrow{a} B_1 \lvert[G]\rvert B_2'$
		$B_1 \xrightarrow{a} B_1', B_2 \xrightarrow{a} B_2', a \in G \cup \{\delta\} \vdash$
		$B_1 \lvert[G]\rvert B_2 \xrightarrow{a} B_1' \lvert[G]\rvert B_2'$
instance	P	$P := B, B \xrightarrow{a} B' \vdash P \xrightarrow{a} B'$

Figure 2.1: Basic LOTOS syntax and semantics

using the structure of the semantics from figure 2.1. The main ideas can easily be expressed using the following decomposition rules :

Inaction ($B =$ **stop**). In this case B is decomposed in $T_1(B) =$ **stop** and $T_2(B) =$ **stop**.

Termination ($B =$ **exit**). In this case B is decomposed in $T_1(B) =$ **exit** and $T_2(B) =$ **exit**.

Action Prefix ($B = a; B'$) : suppose $a \in A_1$, then B is decomposed in $T_1(B) = a$; *sync*!; $T_1(B')$ and $T_2(B) =$ *sync*!; $T_2(B')$

Choice ($B = B_1 [] B_2$): if there is no global choice (e.g. $B_1 = a; B_1'$, $B_2 = b; B_2'$, with $a \in A_1, b \in A_2$), then B is decomposed in $T_1(B) = T_1(B_1) [] T_1(B_2)$, $T_2(B) = T_2(B_1) [] T_2(B_2)$.

Parallel ($B = B_1 \lvert[G]\rvert B_2$) : B is decomposed in $T_1(B) = T_1(B_1) \lvert[G]\rvert T_1(B_2)$ and $T_2(B) = T_2(B_1) \lvert[G]\rvert T_2(B_2)$

For the other semantic rules, similar decompositions can be given. The hardest case seems to be the choice, where global choices are hard to distribute. A possible solution to this problem is proposed in [Lan90].

Note that LOTOS is only a notational vehicle, and that the same problem has also been studied on various types of labeled transition systems ([CMT99, Mor99, SEM03]).

The main criticism on these works, from an industrial point of view, is

that in these formalisms, contrary to programming languages, the notion of assignment to variables does not exist. Other solutions had therefore to be found.

2.1.2 Unity

In a high level framework, Chandy and Misra have proposed the Unity approach [KMC88] to model and design asynchronous or synchronous parallel programs. Let us recall that the principle in Unity, similar to the one proposed by the B method [Abr96], is to use a design modeling language together with a proof system to provide, through several design decisions, correct parallel programs. Central in this framework is the separation of the concerns of program development and the physical architecture on which it is implemented.

unity_program	::=	**"PROG"** prog_name **"READ"** variables_set **"WRITE"** variables_set **"INIT"** state_predicate **"ASSIGN"** actions
actions	:: =	action | action **"[]"** actions | **"[]"** i : quantification$_i$: action$_i$
action	::=	assignment | guarded_action
assignment	::=	variables_list **":="** expr_list
variables_list	::=	var (**","** var)*)
expr_list	::=	expr (**","** expr) *
guarded_action	::=	**"IF"** expr **"THEN"** action

Figure 2.2: Part of Unity Syntax

The program development phase provides the specification of the Unity program in itself using guarded, multiple assignment statements. A part of the syntax for Unity programs can be found in BNF notation [ML87] in figure 2.2. The syntactical elements found in a Unity program are the following :

- **prog** states the name of the Unity program that is specified.
- **read** and **write** contains the declaration of the variables used in the program. Variables may be in both **read** and **write** declarations. Variables declared as **read** are variables governed by the environment (which updates their values), while variables declared as **write** are read by the environment.

- **init** expresses a state-predicate, specifying the set of possible initial values for the variables of the Unity program.

- **assign** declares a set of actions separated by the symbol "[]". The quantified action expression can be used to abbreviate a set of actions. The index i in such an expression is instantiated for each element in the range expressed by quantification$_i$. Each instance of i results in an instance of action$_i$. An example of such an quantified action is : `[] i : 0 ≤ i < 10: A[i] := 0`, which generates a set of ten actions that assign 0 to the first 10 elements of some array A.

```
PROG  Example
READ  {a,x,y}
WRITE {x,y}
INIT  true
ASSIGN
    if a  = 0 then x := 1;
[]  if a != 0 then x := 1;
[]  if x != 0 then y, x := y+1, 0
```

Figure 2.3: A Unity example program

The execution of a Unity program starts from an initial state, which satisfies the init predicate. Next, non-deterministically, one action is chosen and executed in an atomic manner, and the process is repeated. Each action is infinitely often scheduled for execution, and hence cannot be ignored forever. For illustration, consider the Unity program in figure 2.3. The initial condition in this program is missing, so any state can be an initial state. As far as Unity is concerned, the actions of this program can be implemented sequentially, fully parallel or anything in between, as long as the atomicity and the fairness condition of Unity are met. One can easily see that during the execution of this program eventually $x = 0$ will hold, and that for any C eventually $y > C$ will hold. Additionally, at any time $y \geq y_0$, where y_0 denotes the initial value of y.

During the second phase, starting from a Unity program, a *mapping* to an architecture is used to describe a possible implementation. Various possibilities are proposed to implement a Unity program on a distributed architecture. The program variables can be seen as shared variables or communication can be done through FIFO channels. In both cases, a protocol must be explicitly given to preserve the data integrity or synchronize the execution flow.

2.1.3 Synchronous languages

On the programming language side, the most relevant works on automated distribution of reactive systems have been done in the domain of synchronous languages such as ESTEREL [BG92], Lustre [CPHP87] and Signal [LGLL91], which answered questions on how to specify controllers in a natural and semantically well defined way. We concentrate on ESTEREL, since it is the language best fit for our application domain. The reader should keep in mind that the situation for the other synchronous languages is very similar.

Informally, all synchronous languages make the assumption that time is a discrete sequence of instants. Instants take place either periodically or when the environment changes its state (when observable events are generated by the environment). When an instant takes place, the system is provided with a snapshot of the environment, and computes a reaction based on the local state of the system in addition with the current state of the environment. The synchronous hypothesis, which is at the basis of all synchronous languages, states that such instants take no time. When an instant takes place, the system instantaneously commits the computed reaction back to the environment.

Informal presentation

We first give an informal overview of the ESTEREL language, before going into its semantics.

Modules A module is the programming unit in ESTEREL. It has the following shape :

`module` *name* `:`
    interface declaration
    statement
`end module`

Modules can be combined together using the `run` statement which copies the body of a module into another module (no recursion is allowed). The interface declaration may consist of (1) data objects, (2) used functions, procedures and tasks and (3) signals and sensors.

Data Data objects are either primitive or user-defined, and are global to the program. Primitive data can be of one of the following types : `boolean`, `integer`, `float`, `double` and `string`, with the usual operators defined for each type. User defined types are handled as abstract types by ESTEREL.

They may find their use when combining an ESTEREL program with external (e.g. C) code through the use of `functions`, `procedures` and `tasks`. Such external codes for functions, procedures and tasks are expected to have no side-effects, and must return within a fixed amount of time.

There is a major difference between variables and signals in ESTEREL. Since many successive actions may take place at a same instant, variables can hold various values successively at the same instant. This is not possible for signals: signals are either present or absent at a given instant, and remain so during the entire instant.

Signals and sensors Interface signals can be `input`, `output` or `input-output`. The presence of input signals is dictated by the environment, while the presence of output signals is controlled by the program. Input-output signals are controlled both by the program and by the environment. Signals can be *pure* (they are either present or absent during the entire instant) or *valued* signals (when present, they carry a value of a primitive type). Valued signals can be seen as a set of pure signals, one for each possible primitive value. Sensors are used like valued signals but they are always present. Their value remains constant inside an instant, but may change at each instant, outside the control of the program.

Expressions Data expressions are specified as usual, with one extra operator : `?`, which takes the value of a valued `signal`.

Signal expressions can be built using `signal` identifiers, which hold a boolean value indicating its presence or absence, and the usual boolean operators. They are mainly used in `present` testing of signals.

ESTEREL **statements** Since ESTEREL is a synchronous language, some common statements may have a special and thus counter intuitive meaning. The most important notion in ESTEREL is the one that expresses instantaneous execution. When a statement is started on time t and ends on time t', it is said to be instantaneous if $t = t'$, and will thus start and end in the same *instant*. Statements may take time, when $t' > t$, while they execute during several *instants*. Between two *instants*, the environment may react. An ESTEREL program makes the assumption that the environment cannot change state during a given *instant*.

Basic statements There are three basic statements. `nothing` : terminates instantaneously when started. `pause` : pauses the execution for 1 instant. `halt` : pauses forever, and never terminates.

Signal Emission The following instructions make a signal `S` present in the current instant, and terminate instantaneously :
`emit S` (for pure signals) and `emit S(`e`)` for valued signals

Sequencing Sequencing is expressed using the `;` operator as in `p; q`. p is immediately started, and if p terminates instantaneously, q is started in the same instant.

Looping The basic construction of a loop is as follows : `loop` p `end loop`. p is immediately started, and when its execution is ended, it is immediately started anew. Remark that ESTEREL does not allow p to be able to terminate instantaneously when started : it should take at least one instant to complete.

Present signal testing The `present` statement tests for the presence or absence of a signal, and instantaneously transfers control flow to one of the specified branches :
`present` S `then` p `else` q `end present`

Data test The `if` operates on boolean expressions, transfers the control flow instantaneously to one of its branches, and takes the common form :
`if` e `then` p `else` q `end if`

Traps Traps may be seen as exception handling mechanisms :
`trap T in` p `end trap`
`trap T in` p `handle T do` q `end trap`
Upon execution, p is immediately executed. If p ends its execution, so does the `trap` statement. If p executes an `exit T` statement, then the `trap` statement immediately terminates, aborting p. In that case, the second statement specifies that q is instantaneously started. An `exit` statement can only be used inside a `trap` statement. When a `trap` statement contains another `trap` statement with the same `T`, the innermost `trap` will be triggered by an `exit` statement.

Parallel statement The parallel operator allows to combine several statements in synchronous parallelism. Signals emitted by one of the branches are immediately broadcast to all other branches. When a parallel statement is started, all of its branches are instantaneously started. It terminates when all of its branches are terminated, or when at least one of the branches executes an `exit` statement, aborting all branches at that time. Variables can

only be shared among parallel branches if they are read only. Only signals may be shared without any restriction.

p || *q* || *r* || ...

Module instantiation The `run` *M* statement may be seen as the abbreviation for the body of *M*. This allows to combine several modules together. We do not discuss the rather expressive mechanism offered by ESTEREL that allows one to rename signals and variables present in *M* in order to achieve modularity. The interested reader may refer to [Ber00] for more details.

Task Execution The use of external code can be expressed in ESTEREL by the `call` statement. Code that is used in this way is supposed to be instantaneous. In many cases, this is too restrictive. `Task`s allow to start an asynchronous execution of external code that takes time to complete. The completion of such code is then indicated to the ESTEREL program by a `return` (possibly valued) signal associated with the `task`.

Small example

Before going into the formal semantics, consider the example (taken from [Ber98]) in figure 2.4, which has the following behavior : *Emit an output O as soon as two inputs A and B have occurred. Reset this behavior each time the input R occurs.*

The outer loop in this example specifies that the behavior of its body is aborted each time an input `R` occurs. Remark that the parallel operator is used to await both the `A` and `B` inputs. The parallel instruction terminates immediately when both branches terminate, and the output `O` will therefore occur on the exact same time that the last one of the two inputs `A` or `B` is present.

```
module ABRO:
input A, B, R;
output O;
loop
    [ await A || await B ];
    emit O
each R
end module
```

Figure 2.4: A small Esterel example

Formal presentation

We now explain in detail the semantics of Pure ESTEREL. The syntax in BNF form for a Pure ESTEREL program is presented in figure 2.5. The annotations on the right indicate terse syntax, which is used to abbreviate the semantic rules. Pure ESTEREL is the subset of ESTEREL obtained by removing variables, data test, valued signals, tasks and module instantiation from full ESTEREL. The Pure ESTEREL kernel contains the very basics of ESTEREL, allowing to build a program that is semantically equivalent to any other plain ESTEREL program.

Esterel_Prog	::=	instr_list	
instr_list	::=	instr { **";"** instr }	$p\ ;\ q$
instr	::=	**"nothing"**	0
		| **"emit"** S	$!s$
		| **"pause"**	1
		| **"present"** S **"then"** instr_list **"else"** instr_list **"end"**	$?s, p, q$
		| **"suspend"** instr_list **"when"** S	$s \supset p$
		| **"loop"** instr_list **"end"**	$p*$
		| instr_list **"||"** instr_list	$p \| q$
		| instr_list **"trap"** T **"in"** instr_list **"end"**	$\{p\}$ and $\uparrow p$
		| **"exit"** T	k with $k \geq 2$
		| **"signal"** S **"in"** instr_list **"end"**	$p \backslash s$

Figure 2.5: Pure ESTEREL kernel

In ESTEREL, the state of the environment is encoded by the presence or absence of signals. Each signal is shared between the system and the environment. All signals can be manipulated both by the environment and the system. During an instant, a signal is either present or absent. We will see that this principle causes some semantic difficulties, including non-deterministic or incoherent semantics, which are common to all synchronous languages.

The behavioral semantics presented here formalizes a reaction of a program P as a *behavioral transition* of the form

$$P \xrightarrow[I]{O} P'$$

where I and O are input and output events. P' is the remaining of P after one reaction to the input event I. An event is the assignment of

each signal to $\{+,-\}$ (present or absent). Reactions are computed using an auxiliary *statement transition* relation, of the following form :

$$p \xrightarrow[E]{E',k} p'$$

where E is an event defining the presence/absence of all signals in the scope of p, E' defines all emitted signals in the reaction and k is the completion code returned by p. Since instances take no time, certain reactions, that happen inside an instant, will terminate instantaneously. Such reactions are indicated with $k = 0$. Some reactions take time, and *end* an instant. Such reactions have $k > 0$.

$$P \xrightarrow[I]{O} P' \text{ iff } p \xrightarrow[I \cup O]{O,k} p' \text{ for some k}$$

The program P is logically reactive (resp. logically deterministic) w.r.t. I if there exists at least (resp. at most) one program transition $P \xrightarrow[I]{O} P'$. It is logically correct if it is logically reactive and logically deterministic.

Essential rules from the behavioral semantics are :

$$k \xrightarrow[E]{\phi,k} 0 \qquad \textit{(compl)}$$

$$!s \xrightarrow[E]{\{s^+\},0} 0 \qquad \textit{(emit)}$$

$$\frac{s^+ \in E, p \xrightarrow[E]{E',k} p'}{s?p,q \xrightarrow[E]{E',k} p'} \qquad \textit{(present+)}$$

$$\frac{s^- \in E, q \xrightarrow[E]{E',k} q'}{s?p,q \xrightarrow[E]{E',k} q'} \qquad \textit{(present-)}$$

$$\frac{p \xrightarrow[E]{E',k} p' \quad k \neq 0}{p;q \xrightarrow[E]{E',k} p';q} \qquad \textit{(seq1)}$$

$$\frac{p \xrightarrow[E]{E',0} p' \quad q \xrightarrow[E]{F',l} q'}{p;q \xrightarrow[E]{E' \cup F',l} q'} \qquad \textit{(seq2)}$$

$$\frac{p \xrightarrow[E]{E',k} p' \quad q \xrightarrow[E]{F',l} q'}{p|q \xrightarrow[E]{E' \cup F', max(k,l)} p'|q'} \qquad \textit{(parallel)}$$

- The *compl* rule makes 0 complete with code 0, indicating instantaneous termination. Since 1 is the shorthand for pause, termination code 1 will indicate pause.
- The *emit* rule inserts s^+ into the signals emitted by the reaction, making the signal s present.
- The *present+/present-* rules redirect control flow instantaneously to the correct branch, depending on the presence of s.
- The sequence rules make instant execution of several instructions possible in one transition since control is directly forwarded from p to q if p immediately terminates (*seq2*). If p pauses (or exits a trap), completion code $k = 1$ (or $k \geq 2$), so does the sequential statement (*seq1*).
- The *parallel* rule makes p and q advance one step synchronously. They use the same environment E which expresses the broadcast of the common signals. The $max(k, l)$ termination code in the *parallel* rule is a technical subtlety used in conjunction with traps. Remark however that if both p and q terminate instantaneously (completion code 0), the parallel statement does the same. If p or q pauses (completion code 1), so does the parallel statement.

Due to the instantaneous execution of some instructions, some programs may have no meaningful execution, or have several executions for the same input event. This situation is common to all synchronous languages, and can be formalized as follows. Given a program and a fixed input event (defining the presence and absence of all input signals), then control flow is defined, and each `emit` statement is either executed or not in the current instant. A global status, i.e. a status for each signal of the program respecting the given input event is globally coherent iff at least one `emit` statement is executed for each signal assumed present and no `emit` statement is executed for each signal assumed absent.

Two obvious logically incorrect programs are represent in figure 2.6. Module P1 (fig 2.6.a) is not logically reactive (informally, the program states

```
module P1:
output O;
present O then
   nothing
   else emit O
end present
end module
```

(a)

```
module P2:
output O;
present O then
   emit O
end present
end module
```

(b)

Figure 2.6: Logically incorrect programs

that if O is present, than O is not present) while module P2 (fig 2.6.b) is not deterministic (for a given instant, O might be present or not).

Clearly, deciding logical correctness is not efficiently feasible, the general case requires an exhaustive trial of all input configurations. Constructive semantics, a more efficient (and thus more restrictive) approach is therefore currently used in the ESTEREL v5 compiler. We don't discuss constructive semantics here; for an in depth study we refer to [Ber99].

A program P is strongly deterministic for an input event I if it is reactive and deterministic for this event and if, furthermore, there exists a unique derivation for the unique transition $P \xrightarrow[I]{O} P'$. The proof of this unique transition is hard due to the fact that one has to search for an output event O enabling $p \xrightarrow[I \cup O]{O,k} p'$.

An important benefit of the synchronous approach is that the use of the synchronous broadcast in the parallel composition of processes (cfr. the semantics of the *parallel* rule), allows to generate sequential code. The generated code is therefore single threaded and deterministic [Edw00].

The distribution of ESTEREL is studied in [Gir94]. The distribution operates on the sequential automaton code. (1) It duplicates the code on all sites, (2) Based on a unique location for each variable, it removes code that does not apply to a given site, (3) data dependencies are fulfilled via FIFO queues between processes that calculate a value and processes that need that value (4) synchronization is inserted through dummy message passing (for example to keep a site that is only producing values in pace).

Unfortunately, because the semantics must be preserved between the central and distributed program, distributed synchronous languages suffer from a performance problem [Gir94] which, in practice, may not be acceptable. We argue that the parallelism used in ESTEREL is too restrictive (too synchronous) to be used in a distributed environment. This is illustrated by

the following abstract example (interface declaration is left aside; `O, O2, O3, ...` are outputs).

```
emit (O); pause; pause; present O then emit(O2) else emit(O3)
||
pause; pause; emit(O)
```

This program uses the synchronous parallel execution (which combines "concurrency" and determinism) of ESTEREL, and therefore never outputs `O3`. But is this a real parallel program ? We argue that it is not, and in many cases the synchrony introduced by the parallel construct of ESTEREL is not needed or wanted.

2.1.4 Motivations for dSL

We are convinced that for a large set of control systems it is not necessary to have synchronism, and therefore it should not be enabled by default, but in rare cases explicitly asked for by the user.

Furthermore, as pointed out in [WWWK94], we argue that the distribution cannot be completely transparent. Especially not in real-time system designs, where network speeds may have a huge impact on the behavior of the system. It is our opinion that the user should be aware of the distribution, and should tell the compiler/distributer where one may safely introduce network communication. If not mentioned, the program should physically be centralized i.e. run without the use of any network primitive. This certainly complicates the user's task, but results in more reliable software. Furthermore, we are convinced that a controller will almost never be randomly distributed, and that components that are working tightly together (using the ESTEREL's `||` synchronism for example) will naturally end up on the same site after distribution (due to the fact that the signals/sensors they use will physically define an indivisible component).

A analogue problem appears in the Unity framework, since the programmer has no control on the synchronization between the different Unity instructions. Each time the target architecture is modified (e.g. adding or removing processor(s), moving variable(s), refining the distribution, ...), the Unity program has to be modified at the communication level, to fit this new distribution. Therefore, the Unity program must be designed with a desired mapping in mind, which has a negative impact on the reconfiguration flexibility of applications and the transparency of the physical distribution. Additionally, concerning the Unity approach, we argue that the industry needs a language which is based upon industrial standards. At the design

level, Unity seems more sound than what is proposed with the direct use of programming languages (among which $_{\mathrm{d}}$SL), since a Unity design goes through the correctness proof for the design before doing the implementation. Unfortunately, the industrial control world has not yet integrated this more formal approach to design real systems, and still uses more specialized programming languages defined as industrial standards.

Since the use of synchronism has a too important drawback on the performance of a distributed system, we reject the use of the synchronous product [Mil81] as a default. We do however believe that some form of synchronism should be used, and adopt therefore a semantics based on the asynchronous composition of local instantaneous code and global distributed code. The instantaneous code will be used to handle local events, while the global distributed code will offer a way to perform tasks in a distributed manner.

In what follows, we shortly comment what are the possibilities for the execution of instantaneous and distributed code, and and how the simplicity, efficiency and robustness concerns affect the choice of our execution environment.

2.2 Execution Environment

The local instantaneous code will execute in what is commonly called an Input-Process-Output cycle. Each such cycle performs three tasks : it reads the inputs connected to the environment, next it processes events and finally it updates the outputs to the environment. Such a behavior is also present in the synchronous languages, where the process phase of such a cycle happens during one instant. Hence, the treatment of events is considered instantaneous in $_{\mathrm{d}}$SL. On the other hand, there is a notion of concurrency in $_{\mathrm{d}}$SL. Each process in $_{\mathrm{d}}$SL communicates with the other processes over a communication network, modeled as FIFO queues. We make the hypothesis that these communications do need time to complete, and put no bound on the speed at which these communications happen. This means that the local code, for a given component, must be able to run without any synchronization that would make it wait on other components. This asynchronous composition has the advantage that the failure of one site does not introduce deadlocks in *atomic* code on other sites.

The *sequential* code, on the other hand, can be executed in a totally distributed and cooperative manner. These assumptions, of course, imply some restrictions on code and have consequences on the way data values are transmitted between distributed processes.

2.2.1 Distributed Shared Memory

For the distributed code, several models of execution are proposed in the literature. These models can be divided into two sets based on the way they achieve data locality : either move the data to the executing processor, or move the execution to the processor which has the data in memory. Many systems based on the first solution, such as Distributed Shared Memory systems [NL91], have been studied. The principle in these systems is to offer a virtual memory where each address location is accessible on all (distributed) processors. The distributed memory system then controls the accesses to the virtual memory by synchronization messages, in order to assure that the system behaves as if all processes are running on a single processor.

These systems, despite offering a transparent distributed environment, suffer from undesirable border effects which are generally problematic in an industrial environment.

The first drawback in these systems is caused by the need to have some memory consistency model [Lam79, Mos93], which expresses how the different processors see the virtual memory at different moments in time, and the need to replicate data [HHG99]. This mainly causes two problems in such systems : thrashing and false sharing. Thrashing is caused when two (or more) processes are competing for exclusive access to a given location in the memory. In that case, synchronization messages must be exchanged between the two processes, resulting in high communication traffic and almost no productivity. False sharing is a problem caused by the granularity of the memory locations in distributed memory systems : several memory locations are often packed together into memory zones. In the case of false sharing, several processes access different variables in the same zone, and since zones must be locked and unlocked as atomic blocks this causes unnecessary network traffic. Because of these problems, distributed memory systems have very variable performance at execution, which makes it hard to predict how the performance will be during execution [NL91].

Secondly, since the data moves around, the supervision of such systems and its error-recovery - both essential features in industrial applications - may become too complicated [MP97] on $_{\mathrm{d}}$SL's target hardware. Indeed, several versions of the same variable may be present at the same time in the system. When one site breaks down, a global consistent state can be reconstructed only when a history of all accesses and updates to all variables is available. The maintenance of such a history requires important communication overhead, and substantial memory use. Both resources are scarce in the $_{\mathrm{d}}$SL environment which is detailed in chapter 5.

2.2.2 Thread Migration

For these reasons, dSL uses the second solution, a concept known as process or thread migration [Esk90] where a thread of execution is halted on one site, its context (local variables and program counter) is sent to another site, where its context is restored and execution continues. Thread migration is known to enable dynamic load distribution, fault tolerance, eased system administration, data access locality and mobile computing [JC02]. However, in our system, all instructions and global variables are statically assigned to the participating sites and thread migration, determined at compile time, is used to obtain data access locality. This has three benefits: (1) following the state of the system is very easy, (2) all communication and synchronization messages can statically be calculated, resulting in a predictable execution, (3) the communication between processes can, in practice, be reduced to very low level mechanisms. However, we lose the benefits of dynamic load balancing and fault tolerance[1] since the migration policy used in dSL is static.

The shape of dSL can thus be synthesized as an imperative language using a hybrid execution scheme composed of two types of code : local or *atomic* instantaneous code, and distributed *sequential* code that executes using statically calculated thread migration.

[1] Fault tolerant dSL systems can be obtained through the additional possibility to test failures in the communications and hardware (see "unknown values" in section 3.1.5); but needs extra coding and the explicit use of redundancies.

Chapter 3

Presentation of dSL

In the previous chapter, we presented the problem of automatic distribution of reactive systems. We showed that existing solutions have shortcomings, with respect to the kind of systems in our framework, and argued for another solution. With these observations in mind, we designed dSL, a new language for the specification of reactive systems.

Because the results in this work are driven by an industrial motivation to solve the problem, we design dSL starting from a language already in use in the industry to specify the behavior of such systems.

We first quickly show how the ancestor of dSL, SL, is used in practice. Next we show how dSL extends SL, and detail its syntax and semantics. We also show some remarkable properties on the semantics of this new language, which can be exploited in the formal verification (see chapter 6) and testing of systems designed with it.

This chapter is based upon two papers. The main ideas from section 3.2 are published in [DeWMM03a]. The results from section 3.4, were also presented in [DeWGMM05] and [Gen04]. The formal semantics and the lattice of distributions presented there, are a result of fruitful discussions with Alexandre Genon and Cédric Meuter, while the technical details of the proof where handled by Alexandre Genon in [Gen04].

3.1 dSL's ancestor SL

dSL stands for distributed Supervision Language, where SL (Supervision Language) is a language currently used in the industry to specify the overall control of industrial processes. However, SL cannot be used to specify the behavior of all the different entities in the system. These entities are of two kinds : on the one hand, there are several *controllers* with a direct link

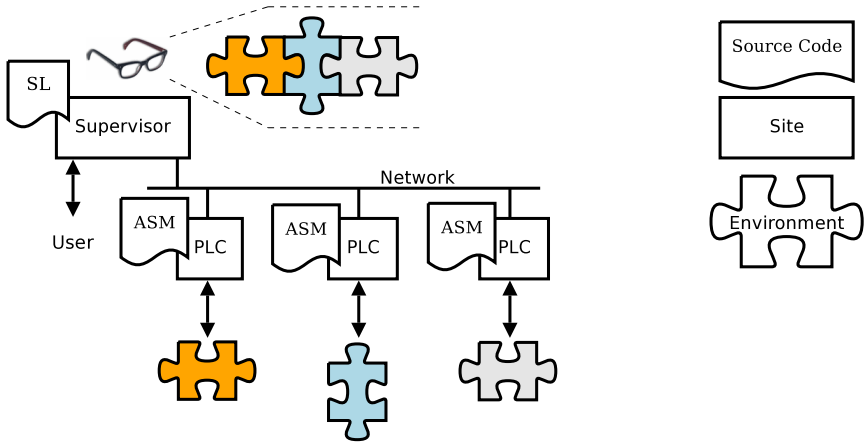

Figure 3.1: SL setup

to the environment, but with a local view, and on the other hand there is a centralized *supervisor* that gathers information about all the controllers. The supervisor has a global view of the entire system and commands the controllers. This situation is depicted in figure 3.1.

The controllers

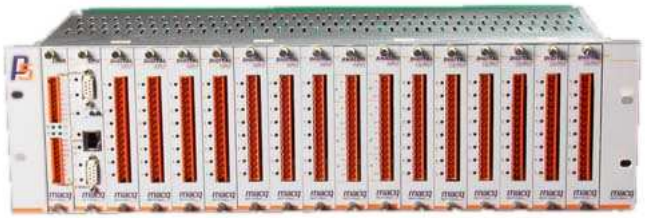

Figure 3.2: A Controller

Each controller (also called Programmable Logic Controller, or PLC in short) is a piece of hardware, as seen in figure 3.2, consisting of a communication interface, processor, memory and Input-Output devices. Each input device can observe the environment, i.e. it can read the status of a set of sensors (monitoring e.g. temperature, speed, angle, position, pressure, ...) while the output devices are able to act upon the environment by forcing some voltage on a set of actuators (controlling e.g. heating elements, engines, valves, ...). The language used to specify the behavior of the con-

trollers is a different one than the one used on the supervisor. Indeed, the controllers are specified in an assembly-like language, with a small instruction set. Instructions include basic arithmetic and PID (for Proportional, Integral, Derivate) control [AH94], allowing fast local response to the environment without the intervention of the supervisor. It is important to notice that the controllers have only a local view of the environment, and can only be told about the state of the entire system through the information given to them by the supervisor.

The supervisor

The supervisor, which typically has a graphical interface to the user, is a high performance server which communicates over a network with the various controllers. The language SL is used to specify the behavior of the supervisor. The idea is that the controllers report their status to the supervisor which is able to construct a global view of the state of the system, and therefore can give the required feedback to the different controllers. In practice this means that the controllers send messages over the network containing the state of their own variables, while the supervisor sends messages to the controllers which have an effect on their behavior.

We will not go into the description of the assembler language used by the controllers, but focus on SL instead. We will see how this language can be extended to dSL, in such a way that it is able to be used as a single source to specify the behavior of the system, without making the difference between the supervisor and the controllers.

The language SL

The language SL, used by Macq Electronique, is an Object Oriented language based on an industrial standard [BMS97], with a syntax much like Pascal. SL has some constructs allowing to handle the specific needs in the case of reactive systems. Appendix A presents SL's syntax. The SL code is interpreted on the supervisor. There is therefore no compiler of any sort for this language.

We briefly present SL's common constructs (in the sense of constructs that are part of most well known imperative languages). We give more attention to the domain specific part of SL next.

An SL program has the following parts : (1) An optional `USES` declaration, which allows to organize programs in separate modules, and will not be discussed any further. (2) Data type declarations. (3) A global statement

list, which is a list of method declarations and event declarations. (4) The initialization code for the SL program.

Data Type Declarations SL basic types include `BOOL`, `SINT`, `USINT`, `DINT`, `UDINT`, `LINT`, `ULINT`, `REAL`, `LREAL`, `CHAR` and `STRING` (resp. boolean, 16-bits signed/unsigned-, 32-bits signed/unsigned-, 64-bits signed/unsigned-integers, 64-bits real numbers, 128-bits real numbers, 8-bits characters and strings). These types can be combined in more complex types using a language construct called `STRUCTURE`[1] which is the equivalent of Pascal's `record`. More dynamic types in SL include pointers, arrays and lists. The other features are not discussed here, since they are not implemented in $_d$SL. SL has some notion of polymorphism using inheritance by defining `STRUCTURE`s that may be `DERIVED` from other `STRUCTURE`s. Only single inheritance is allowed in SL.

Method Declarations Methods are defined on previously defined `STRUCTURE` types. Parameter passing to methods is always by value. Note that methods in SL have no return type or value. Methods have an optional declaration of local variables, and a body which is a list of instructions. Methods defined on `DERIVED` types override methods (with the same name) from the parents and are always virtual. Inside a method, `self` denotes the pointer addressing the object on which a method is called.

Event declarations, also called `WHEN` declarations, are discussed in detail further on.

Initialization Code The initialization code is marked by the keyword `PROGRAM` and contains an optional declaration of local variables and a list of instructions.

Instructions Next to the common instructions (assignment, `IF`, `WHILE`, `REPEAT` `FOR` and method calls), some specific statements exists in SL, which will not be discussed any further here.

3.1.1 Static memory

There is no dynamic memory allocation mechanism in SL (i.e. there is no equivalent for the `new` and `dispose` operators in Pascal). This means that the memory needed during execution, except for the call stack that

[1] In the parser, developed at the ULB, we decided to change `STRUCTURE` into `CLASS`, for obvious reasons

may be unbounded due to recursion, is statically known. Global objects are allocated once and stay live during the entire execution. SL does know pointers, but there is no pointer arithmetic as in C.

3.1.2 SL Variables

Internal variables

In SL, one can make the difference between four sets of variables. As is usual in all imperative programming languages, variables are either local or global (i.e. variables with a lifetime throughout the whole program, or variables that are alive only inside the structure of a compound construct such as a method). These common variables will be called internal variables, and are readable and writable in their scope.

External variables

In addition to these internal variables, SL contains two sets of external variables, which are global. These external variables are either input or output variables.

An input variable is linked to a sensor, i.e. it is linked to a sensor which is connected on a certain controller. Such a variable is only readable by the program. The value of such a variable is linked to the physical state of the sensor on the controller to which it is associated, and the network is used to communicate updates of the observed values to the supervisor. For a temperature sensor for example, the input variable `temp` linked to that sensor will contain an integer reflecting the temperature of the environment observed by a controller which is wired to that sensor. The values read by the controller are communicated to the supervisor which updates the variable `temp` with the communicated values.

An output variable is linked through a connected controller to an actuator. Such a variable is readable and writable in the SL program. When the program changes the value of such a variable, a message is sent over the network, to make the connected controller apply the new value to the actuator. In this way, the SL program can influence the environment. For an engine for example, the output variable `speed` linked to an engine on a controller, can be set to a certain value, expressing the rounds per minute at which the engine will turn in the environment.

External variables may also be linked to a graphical user interface. Input variables, for example, may be linked to a scroll-bar widget on a graphical interface. When the user changes the position of the scroll-bar's cursor, a

message is generated with the new value, which will update the value of that particular SL variable. Output variables may change the form or color of polygons on the graphical user interface. Macq Electronique's *Obviews* software allows to specify such interactive graphical user interfaces.

In practice however, no syntactical constructs make the difference between internal and external variables. These differences, and the mapping between each external variable and its sensor/actuator on a certain controller are specified in a database that is read by the SL's execution environment. The controllers have to be manually programmed, in addition to their local control, in the dedicated assembler-like language to communicate variables to the supervisor. The supervisor interprets these messages automatically, using the database.

It is not clear in SL exactly *when* the external variables are updated, i.e. there seems to be no hard constraints on the time between the update of an output variable and the physical change of voltages on an actuator, nor between the time a sensor *sees* a change, and the time the linked input variable changes its value on the supervisor.

3.1.3 The `WHEN` construct and assignment semantics

Since SL is used to handle reactive systems, the language contains compound instructions allowing to react on changes from the environment. Such observable changes that require a certain reaction from the system are called events. In SL, an event is directly associated with its reaction, through the means of a compound instruction called a `WHEN`.
The syntactical structure of a `WHEN` is as follows :

`WHEN` *condition* `THEN` *instruction_list* `END_WHEN`

A `WHEN` contains two parts : (1) a condition, which expresses the event to observe, and (2) a body, which is a list of instructions that express the reaction to the event.

Informally, each such a `WHEN` is continuously monitored, and when a rising edge occurs in the condition (a change in the condition from `FALSE` to `TRUE`), the associated instructions are executed. To stress that there has to be a rising edge in the evaluation of the condition, we sometimes use the term triggering condition. A `WHEN` is declared globally, the variables in its triggering condition can be external or internal global variables and the global variables referenced to in the condition must be statically computable (i.e. pointers, list or array access are not allowed).

A typical example of a `WHEN` is :

`WHEN` door.opened `THEN` door.engine.command := `FALSE`; `END_WHEN`

In this example, the engine controlling an automatic door is shut down when the door reaches its opened position. Notice that the rising edge semantics will only shut down the engine when the door changes from not opened to opened.

Since the conditions in the `WHEN`s are continuously monitored, this implies that the semantics of the assignment instructions (`lhs := rhs;`) in SL may have side effects. Indeed, in addition to the expected behavior where the left hand side is updated with the evaluation of the right hand side, all `WHEN`s are checked in order of appearance in the program text immediately after the assignment and triggered, in that same order, if needed. The fact that conditions may be expressed using internal global variables, may cause problematic behavior: consider the following `WHEN`,

`WHEN` x `THEN` x := `FALSE`; x := `TRUE`; y := 1; `END_WHEN`

where x is a global writable variable. With SL's semantics of assignment, this program clearly has an infinite sequence of internal events, all of which trigger this `WHEN` once x changes from `FALSE` to `TRUE`. Moreover, if no cut is done in these successive triggerings, the assignment `y := 1;` will never be reached.

In the implementation of SL, however, this does not seem to bother the industrials, since a stack is used to handle the recursive behaviors of the `WHEN`s, and infinite cycles are interrupted by the limited amount of memory on the stack. They will argue that such a program is incorrect. Notice that the problem of infinite recursion of triggering conditions in SL is closely related to the termination problems in active databases [KU96].

Any desired behavior in a SL program is written using this `WHEN` construct, often resulting in a sequence of `WHEN`s that encode a finite state machine. Programs written this way are hard to understand and not easy to maintain. We will see how dSL allows the programmer to avoid having to use the following programming style :

`WHEN` state = 1 `AND` temp > 30 `THEN` heater := 10; state := 2; `END_WHEN`
`WHEN` state = 2 `AND` temp > 60 `THEN` heater := 1; state := 1; `END_WHEN`

3.1.4 The `WHEN IN` construct

SL, is an Object Oriented language. In order to specify an event on all object instances from the same type, one can use the `WHEN IN` construct :

`WHEN IN` *type_definition condition* `THEN` *instruction_list* `END_WHEN`

Since SL has no dynamic object instantiation, this is syntactical sugar, but can be very helpful for the programmer. If for example, several automatic doors have to be controlled, the following `WHEN IN` could be used, instead of specifying a `WHEN` for every single door :

`WHEN IN` Door self.opened `THEN` self.engine.command := `FALSE`; `END_WHEN`

3.1.5 The `UNKNOWN` value

All basic types in SL `BOOL, INT, REAL, ...` range over the expected domains (cfr. section 3.1), for which we say that a variable contains a known value. In addition to these values, a special value `UNKNOWN` is added, in which case we say that a variable contains an unknown value. Unknown values model the hardware failure of the physical device to which an external variable is linked, or a breakup in the communications between the supervisor and the controller to which a linked external variable is attached. A special instruction `IS_UNKNOWN` allows to check if a variable contains a known or unknown value.

Unknown values are propagated through expression evaluation, for example if `x := y + 3;` where `y` carries an unknown value, `x` will be updated with an unknown value. See page 49 for a formal definition, and the ways in which `UNKNOWN` values are or are not propagated.

As shown in figure 3.3 this allows to construct more robust programs. In this figure, a sensor is duplicated in order to ensure a correct behavior in case of hardware failures. The variable `handled` ensures that only one of both (the first in order of appearance) is triggered when both sensors are functioning correctly. Note that the body of a `WHEN` whose condition evaluates to `UNKNOWN` is not executed.

```
WHEN temp1 > 30 AND          WHEN temp2 > 30 AND
     NOT handled THEN             NOT handled THEN
    handled := TRUE;             handled := TRUE;
    ...                          ...
END_WHEN                     END_WHEN

       WHEN IS_UNKNOWN(temp1) AND
            IS_UNKNOWN(temp2) THEN
           alarm := TRUE;
           ...
       END_WHEN
```

Figure 3.3: Fault tolerance in $_D$SL with `UNKNOWN`.

3.1.6 Looping semantics

A program written in SL has no real entry point, except for a list of instructions used for initialization purposes. Once these instructions are terminated, the interpreter takes over control, and waits for messages from the controllers and the user interface. Incoming messages are buffered into a FIFO queue, and handled one by one.

Each message is a couple of the form (x, v) where x is a variable and v its updated value. The interpreter unpacks the message, and executes the instruction $x := v;$, which may cause the triggering of several `WHEN`s as explained before. Each such a treatment is called a message treatment cycle.

Notice that if, for some reason, the time needed for a message treatment cycle to complete is not finite (infinite `WHEN` triggering, infinite loops, unbounded recursive method calls, ...) no more messages will be handled by the interpreter, and the system will be in a livelock state. To make sure that the system remains responsive, a hardware countdown clock is set to a certain amount of time at the beginning of each message treatment cycle. When the countdown reaches 0, a hardware alarm is issued, and the message treatment cycle is interrupted. It is however unclear what will be the state of the system after such a problematic behavior.

3.2 From SL to $_d$SL

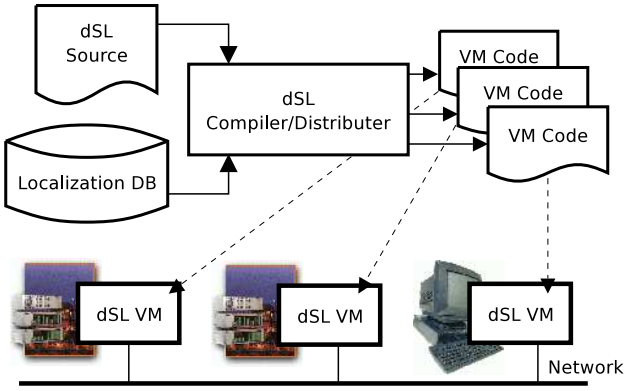

Figure 3.4: $_d$SL organigram

As argued before, $_d$SL has chosen to extend SL, in such a way that it provides the programmer with transparent distribution. Therefore, in $_d$SL, there is no difference anymore between controllers and supervisor: both entities are called sites. $_d$SL extends SL in such a way that the programmer

specifies in a single $_d$SL program the behavior of the system, as if all actuators and sensors were attached to a single site (we shall see that some restrictions are imposed to apply this principle). Once the system's behavior is specified, the designer must then fill in a *localization table*, analogue to SL's database with external variable descriptions. This table is used to specify the physical localization (execution sites) of each external variable.

It is then the task of the *compiler-distributer* to assign each instruction to a single site, in order to satisfy the locality constraints given in the localization table (see figure 3.4). Once each global variable and each instruction is correctly localized on a certain site, the distributer can generate an executable code for each site in the system. For technical reasons, which are mainly based on maintainability, this code is interpreted by a $_d$SL virtual machine.

The localization of all global variables (external and internal) is at the center of $_d$SL's automatic distribution mechanism. Internal variables are either global, in which case their localization is statically fixed by the distributer, or local, in which case they can move during execution. Since global variables do not move during execution, the distributer has to ensure that an instruction accessing a global variable is executed on the site of that variable.

$_d$SL is chosen to be backwards compatible with SL, and has therefore all the language elements presented earlier. However, SL's pointers and inheritance are not taken into account by the distributer. The reason for this omission is that these concepts are rarely used in SL programs. We will therefore not discuss these concepts any longer, and will only come back to them as future work in chapter 7.

So what are the syntactical additions and semantics, that make this transparent distribution possible ? We will answer this questions in detail in the next sections, but first we present a small complete $_d$SL example in figure 3.5, that will be used as a running example.

A $_d$SL program contains 5 elements: (1) global variables declarations, including all external variables (2) method definitions (3) when instructions (4) sequence definitions and (5) an initialization.

In the example, a temperature sensor is linked to an input variable `temperature`. A heater is turned on (off) if the temperature is below 0^oC (above 20^oC). The state of the heater (on/off) is controlled by the output variable `heater.state`. Moreover, there are two indicators on a control panel. The first indicator, (linked to the output variable `led`) is used to indicate the state of the heater, and the second (linked to the output variable `alarm`) is updated when the heater has been turned on a certain number

```
01: CLASS Heater
02:    control              : INT;
03:    maintenance, state   : BOOL;
04: END_CLASS
05:
06: GLOBAL_VAR
07:    heater                 : Heater;
08:    temperature, fuel_cost : INT;
09:    alarm, led             : BOOL;
10: END_VAR
11:
12: SEQUENCE set_heater(new_state : BOOL)
13:     heater.state := new_state;
14:     IF heater.state THEN
15:         led       := TRUE;
16:         fuel_cost := fuel_cost + 10;
17:     ELSE
18:         led := FALSE;
19:     END_IF
20:     heater.control := heater.control + 1;
21: END_SEQUENCE
22:
23: WHEN IN Heater (control==1000) THEN // W1
24:     control := 0;
25:     maintenance := TRUE;
26: END_WHEN
27:
28: WHEN heater.maintenance THEN        // W2
29:     alarm := TRUE;
30: END_WHEN
31:
32: WHEN ~temperature < 0 THEN          // W3
33:     IF (NOT heater.maintenance) THEN
34:         LAUNCH set_heater(1);
35:     END_IF
36: END_WHEN
37:
38: WHEN ~temperature > 20 THEN         // W4
39:     IF (NOT heater.maintenance) THEN
40:         LAUNCH set_heater(0);
41:     END_IF
42: END_WHEN
43:
44: PROGRAM
45:     heater.control := 0;
46:     heater.maintenance := FALSE;
47:     LAUNCH set_heater(temperature<0);
48: END_PROGRAM
```

Figure 3.5: A temperature control system in $_{\mathrm{d}}$SL

of times. An additional variable `fuel_cost` estimates the amount of fuel consumed by the heater.

3.2.1 Three syntactical additions

In the previous chapter, we motivated $_d$SL's basic design : it uses a hybrid execution scheme composed of local or *atomic* instantaneous code, and distributed *sequential* code.

All code inside a `WHEN` (the instructions inside its body and the evaluation of the condition) or reachable from a `WHEN` must a priori be *atomic*, and therefore local to a given site.

Sequential code is defined through the use of the `SEQUENCE` construct. The code inside a `METHOD` can be either *atomic* or *sequential* depending on the context in which it is called. If a `METHOD` can be reached from a `WHEN`, then the body of this `METHOD` is assumed to be atomic. It is assumed *sequential* otherwise. Hence, when a `METHOD` is both called from a `SEQUENCE` and from a `WHEN`, it is atomic.

As explained before, $_d$SL rejects the synchronous product, and atomic code can therefore not be distributed. This may result in programs that are not distributable. For instance, without a strong synchronization scheme, the following code, where the external variables `pushbutton` and `lamp` are fixed by the localization table on different sites, is not distributable :

```
WHEN pushbutton THEN lamp := TRUE; END_WHEN
```

Indeed, this snippet of code specifies that the lamp must switch on at the exact same time the button is pressed. With the hypotheses of $_d$SL in mind, this is clearly not feasible if the lamp and the button are on different sites: to implement such a behavior the sites must communicate, and by hypothesis, communication takes time.

In this case, the designer has two options. Obviously, by rewiring the lamp or the button, such that both are on the same site, the program becomes implementable. On the other hand, it might be the case that a communication delay is acceptable, in which case the designer must have a syntactical way to relax the atomic constraints imposed by a `WHEN`. Three syntactical additions have therefore been defined (see figure 3.5) : (1) the `SEQUENCE` keyword, (2) the $\sim$ operator and (3) the `LAUNCH` keyword. We comment these additions in detail in the following sections.

The SEQUENCE keyword

The SEQUENCE keyword can be used to state that a certain list of instructions does not have to be executed on one site, nor in one cycle. Code that is embodied into a SEQUENCE declaration, can therefore be delayed and thus distributed. A typical usage for a sequence is to do a list of actions, each of which may be happening on a different site :

```
SEQUENCE plant_startup()
     pump1.engine    := 10;
     pump2.engine    := 10;
     valve1.pressure := 5
     ...
END_SEQUENCE
```

The initialization of a $_d$SL program is done in the PROGRAM part, and has the same semantics as a SEQUENCE.

Since each instruction may be located on a different site, extra code is generated by the distributer, which makes the execution synchronize between different sites. The execution of a sequence does not happen instantaneously (i.e. in one cycle) since the execution of a SEQUENCE may involve different sites. Note however that each single instruction in a SEQUENCE must be located on a single site, and is executed in an atomic manner.

The local variables in the SEQUENCE are moved along with the sequence during execution, as shown in the following example, where a local variable flag is tested at the end of the sequence, which contains the combined value of the status of three pumps. Note that the alarm and the three pumps all may be on a different site.

```
SEQUENCE plant_check()
     BEGIN_VAR
       flag : BOOL;
     END_VAR
     flag := pump1.heat > 80;
     flag := flag AND pump2.heat > 80;
     flag := flag AND pump3.heat > 80;
     IF flag THEN
        alarm := TRUE;
        ...
     END_IF
END_SEQUENCE
```

Remark that in this example, `flag` may be true even if one of the pumps has cooled down during the time that the `SEQUENCE` reaches the `IF` condition. If the programmer absolutely wants to check the combined exact value of the temperature of the three pumps, the following should be written :

```
SEQUENCE plant_check2()
     BEGIN_VAR
       flag : BOOL;
     END_VAR
     flag := pump1.heat > 80 AND
             pump2.heat > 80 AND pump3.heat > 80;
     IF flag THEN
        alarm := TRUE;
        ...
     END_IF
END_SEQUENCE
```

Since each $_d$SL instruction must be located on a single site, the only way for this program to be distributable is to put the three heating sensors on the same site.

If in addition, the alarm must be raised on the exact same time that the three pumps overheat, the designer should not use a sequence but use the following code, which is only implementable if all heating sensors and the alarm are on the same site:

```
WHEN pump1.heat > 80 AND pump2.heat > 80
     AND pump3.heat > 80
THEN
     alarm := TRUE;
END_WHEN
```

For the programmer's convenience, $_d$SL has one instruction which does not exist in SL, and can only be used inside a `SEQUENCE`. The `WAIT` instruction expresses that a `SEQUENCE`'s execution is paused until a certain condition occurs.

```
SEQUENCE start_engine()
   // Set motor speed to 100
   motor.speed_command := 100;

   // Wait until speed is reached
   WAIT motor.speed_captor = 100;

   // ...
END_SEQUENCE
```

This instruction is syntactical sugar, since it can easily be transformed through the use of `WHEN`, as explained in section 3.6. Note that, in contrast to the rising edge semantics of the triggering condition of a `WHEN`, if the condition of a `WAIT` evaluates to `TRUE` when the instruction is reached, execution continues.

A given `SEQUENCE` can have only one running instance at a time. This restriction is imposed by the limited amount of memory available on $_d$SL's target platform. To ensure that this is the case, for a given `SEQUENCE`, a flag is raised when the `SEQUENCE` is started, and lowered when it reaches the end of the execution. This flag is available to the programmer through the special variable `a_sequence.ENDED` which allows one to check that a certain `SEQUENCE` with the identifier `a_sequence` has ended its execution. The addition of this variable is syntactical sugar. It is introduced for each `SEQUENCE` by the $_d$SL compiler as a global variable residing on the site of the first instruction of the `SEQUENCE`. When a `SEQUENCE` is `LAUNCH`ed while it is still running, the request is ignored.

The use of the $\sim$ operator, presented next, in `SEQUENCE`s is not useful, and is therefore syntactically forbidden.

The implementation details of how a sequence is translated in executable code is explained in chapter 5.

The $\sim$ operator

The $\sim$ operator can be used to break the atomicity constraints induced by the instantaneous handling of events. It works much like the way the supervisor sees the system within the SL setup. There, controllers monitor the environment, and send messages to the supervisor to inform the changes in their local states. This is exactly what does the $\sim$ operator.

When a variable `x` is located on some site and changes its value, that site will send an update message to all sites that use `~x`. These distant sites

can therefore observe `x` with a delay induced by the network. Notice that with the $\sim$ operator, several sites may have different copies of `~x` : suppose site A governs `x`, while B and C both use `~x`. When `x` changes, site A sends update messages to B and C. If B and C do not read the messages at the same time, they will have different copies for `~x`.

Of course, one should be very careful with the $\sim$ operator, since there might be a difference between its value and the real value of `x`, which is the case when the update message is still traveling over the network. But in many practical cases, the $\sim$ operator is very useful. This is the case for most slowly changing physical measures, such as for example observations of evolving temperature, pressure, speed, etc.

Consider the following `WHEN`, where `pump1.temp` and `pump2.speed` are at different sites:

```
WHEN pump1.temp > 80 THEN pump2.speed := 10
END_WHEN
```

In order for this program to become implementable, the designer can use the $\sim$ operator as follows :

```
WHEN ~pump1.temp > 80 THEN pump2.speed := 10
END_WHEN
```

In this case, the entire code of the `WHEN` will be localized on the site where `pump2.speed` is localized, and the site governing `pump1.temp` will send updates when the value of `pump1.temp` changes.

The `LAUNCH` keyword

The `LAUNCH` keyword is the dual of the $\sim$ operator. Instead of operating on data (as does the $\sim$ operator), it operates on code. If we take the previous example again, where `pump1.temp` and `pump2.speed` are at different sites:

```
WHEN pump1.temp > 80 THEN pump2.speed := 10
END_WHEN
```

The designer can use the `LAUNCH` keyword to make this program distributable as follows:

```
METHOD pump::setspeed(s : int)
    self.speed := s;
END_METHOD

WHEN pump1.temp > 80 THEN LAUNCH pump2.setspeed(10);
END_WHEN
```

In this case, a message is sent from the site governing `pump1.temp` to the site governing `pump2.speed`, to update the speed. The `WHEN` will therefore be located on the site of `pump1.temp`.

3.2.2 The `UNKNOWN` value in $_{\mathrm{d}}$SL

The `UNKNOWN` value has also been adopted in $_{\mathrm{d}}$SL. There are two sources of an `UNKNOWN` value in SL. Either it is explicitly introduced using the `UNKNOWN` constant in expressions, or it is produced by the environment to signal a hardware failure.

In $_{\mathrm{d}}$SL, there is an additional source of `UNKNOWN`, due to communication failure. Each site in $_{\mathrm{d}}$SL regularly checks that it is still able to communicate with all other sites. If this is not the case anymore, the execution environment sets all values of tilded variables for those sites to `UNKNOWN`. This can be used (similar to the example given in section 3.1.5) by the programmer to detect errors in communications and allows to build more robust systems.

3.2.3 Dynamic concepts

Two dynamic features are supported in $_{\mathrm{d}}$SL: the use of the reference `self` inside `METHOD`s and the use of arrays. Both features are transformed in such a way that they become *static*, i.e. references to objects through the use of `self` and dynamic array access are replaced by references to statically known objects by the $_{\mathrm{d}}$SL compiler.

The `self` reference in `METHOD`s is removed by the compiler using a technique we call *specialization*, which is discussed in section 5.1.3. Specialization together with a simple code transformation is used to remove the dynamic aspects of arrays. We also discuss this transformation in section 5.1.3.

3.3 $_{\mathrm{d}}$SL Syntax

The complete $_{\mathrm{d}}$SL syntax can be found in BNF form in appendix B.

3.4 $_{\mathrm{d}}$SL Semantics

In this section, we formally describe the $_{\mathrm{d}}$SL semantics. However, in order to keep the semantics simple, we present only a subset of the $_{\mathrm{d}}$SL language. In this subset, called $_{\mathrm{d}}\mathrm{SL}_{\diamond}$, we make the following restrictions: (1) methods are supposed to be inlined, which implies that recursive calls are forbidden; (2) since no recursion is allowed, all variables outside `SEQUENCE`s are considered

to be declared globally; (3) only boolean variables are considered; `METHOD LAUNCH`es and `WAIT` instructions inside `SEQUENCE`s are not considered, since they do not increase the expressiveness of the language and can equivalently be automatically translated into code using `WHEN`s and $\sim$ as explained in section 3.6.
From now on, we will use the following notations:

- $Var(P)$ denotes the set of non tilded variables appearing in the program P. This set is partitioned into Var^{in}(P), Var^{out}(P) and Var^{τ}(P) which correspond respectively to the input, output and internal (i.e. not I/O) variables.
- $Var(i)$ denotes the set of variables appearing in instruction i
- $Var(w)$ denotes the set of non tilded variables appearing in the `WHEN` w.
- $Var(e)$ returns the set of variables (*tilded or not*) appearing in expression e.
- $VarSeq(P)$ returns the set of local variables in the `SEQUENCE`s of P. We suppose that all these variables have a different name.
- $VarSeq(s)$ returns the set of local variables in the `SEQUENCE` identified by s.
- $<_V(P)$ denotes the order in which the variables of a program P are declared. It will be used to determine the order in which the input variables, respectively output variables, will be sampled, respectively updated. We denote $<_V$ if P is clearly identified.
- $Whens(P)$ denotes the set of `WHEN`s appearing in the program P.
- $<_W(P)$ denotes the order in which the `WHEN`s of program P are declared. This order will be used to determine the order in which the `WHEN`s will be processed.
- $Cond(w)$ denotes the triggering condition of the `WHEN` w.
- $Body(w)$ denotes the body of the `WHEN` w
- $OldCond(W) = \{old\_cond_w | w \in W \subseteq Whens(P)\}$ where for all $w \in Whens(P)$, old_cond_w is a new *id* corresponding to the previous evaluation of $Cond(w)$. $OldCond(P) = OldCond(Whens(P))$

- $\tilde{X}$ denotes the set of tilded variables corresponding to X, $\tilde{X} = \{\tilde{x}|x \in X\}$
- $\mathcal{S}(s)$ returns the body of the `SEQUENCE` identified by s
- $InstrSeq(P)$ denotes the set of all instructions appearing in the `SEQUENCE`s of P

Moreover, in order to define the $_{\mathrm{d}}\mathrm{SL}_{\diamond}$ semantics, extended instructions are added to the language which will be used to describe some internal treatment:

- `INPUT` *id*, modeling the sampling of the input variable *id*,
- `OUTPUT` *id*, modeling the update of the output variable *id*,
- `BCAST` *id*, modeling the broadcast of the variable *id* to all execution sites,
- `MSG`, modeling the treatment of messages from the FIFO channel.

Informally, the behavior of a $_{\mathrm{d}}\mathrm{SL}_{\diamond}$ program can be seen as the parallel composition of n processes, one for each site, communicating with the environment to control. Each process P_i governs a set V_i of variables and communicates with the other processes through FIFO channels. In particular, these FIFO channels allow to update the value of the $\sim$ variables, which are the local values of the distant variables.

The execution of each process i is an infinite loop of cycles called *Input-Process-Output* cycle because it contains three phases. (1) *Input* : variables linked to inputs change their value according to the physical state of the device they are attached to, (2) *Process* : events are triggered, incoming messages are processed and `SEQUENCE`s are executed; and (3) *Output* : variables linked to the outputs update the physical state of the devices they are connected to.

3.4.1 Definition of Distribution

As we have already seen, the main difference between $_{\mathrm{d}}\mathrm{SL}_{\diamond}$ and any common imperative programming language results from its distributed characteristics. Indeed the behavior of a $_{\mathrm{d}}\mathrm{SL}_{\diamond}$ program depends on the distribution of its variables. However, the *maximal distribution* of a $_{\mathrm{d}}\mathrm{SL}_{\diamond}$ program can be defined. It expresses the most liberal configuration on which the $_{\mathrm{d}}\mathrm{SL}_{\diamond}$ program could ever run. Indeed, due to *atomic* code, some instructions and

hence variables must be kept on a single site; but in the case of sequential code, the code may be distributed. We will show that the possible behaviors of a program with this *maximal distribution* includes the behavior of that program with any other distribution. Therefore, verifying safety properties on this distribution will induce safety for any other distribution. In what follows, we only consider *well-formed* $_{\text{d}}$SL programs where each used variable has been properly defined and each `WHEN` uses at least one global variable.

Before we can define the *maximal distribution*, we first need to define what a distribution is. Intuitively, a distribution of a well-formed $_{\text{d}}\text{SL}_\diamond$ program is a partition of the set of its variables respecting the atomic constraints imposed by `WHEN`s. That is, if two variables appear un-tilded in the same `WHEN`, they must be localized on the same execution site.

Definition 1 (Distribution of a well-formed $_{\text{d}}\text{SL}_\diamond$ program) *A distribution of a well-formed $_d SL_\diamond$ program P is a partition $D = \{V_1, V_2, ..., V_n\}$ of $Var(P)$ such that*

$$\forall w \in Whens(P) \exists V \in D, Var(w) \subseteq V$$
$$\wedge \forall i \in InstrSeq(P) \exists V \in D, Var(i) \subseteq V$$

We note $\mathcal{D}_P$ the set of all distributions of P. ♦

In order to define the maximal distribution, we need a way to compare two distributions. Therefore, we introduce a partial order relation defining a hierarchy of distribution.

Definition 2 (Distribution hierarchy) *Let $D = \{V_1, V_2, ..., V_k\}$ and $D' = \{V'_1, V'_2, ..., V'_l\}$ be two distributions of a $_d SL_\diamond$ program P. We say that distribution D' is a refinement of distribution D, noted $D \preceq D'$, if*

$$\forall V' \in D' \exists V \in D \cdot V' \subseteq V$$

♦

Theorem 1 (Distribution Lattice) *D and $\preceq$ define a lattice of distributions.* ■

Proof. The lattice defined by $\preceq$ has a trivial least element $D_{min} = \{Var(P)\}$, which corresponds to the distribution where all variables are in one partition. To show that there exists a greatest element D_{max}, we proceed by contradiction. Suppose therefore that $D = \{V_1, ..., V_l\}$ and $D' = \{V'_1, ..., V'_k\}$ both are maximal elements with $(D \neq D')$. We have $D \not\preceq D'$ and $D' \not\preceq D$. In that case, we have

$$\exists i \in [1..l] : \forall j \in [1..k] : V_i \not\subseteq V'_j \qquad (1)$$

Let us construct a new distribution D'' using V_i as follows : $D'' = \{V'_1 \setminus V_i, V'_2 \setminus V_i, \ldots, V'_l \setminus V_i, V_i\}$.
Because of (1), there is no empty partition in D'', and since V_i is a partition of D, we know that D'' is a distribution.
Notice that $D' \preceq D''$, which is in contradiction with the fact that D' is maximal. □

In section 3.5.1, we will use this property to show that the semantics, which is influenced by the distribution, also forms a lattice.
We can now formally define the *maximal distribution* of a well-formed $_{\mathrm{d}}\mathrm{SL}_{\diamondsuit}$ program, which is the the most refined distribution.

Definition 3 (Maximal distribution of a $_{\mathrm{d}}\mathbf{SL}_{\diamondsuit}$ program) *The maximal distribution of a well-formed $_d$SL$_\diamondsuit$ program P is the distribution D_{max} such that*

$$\nexists D \neq D_{max} : D_{max} \preceq D$$

♦

3.4.2 Preliminary definitions

Before we can define the structural operational semantics of a $_{\mathrm{d}}\mathrm{SL}_{\diamondsuit}$ program, we need some preliminary definitions. First, the semantics will be defined in terms of a labeled transition system. We briefly recall this notion.

Definition 4 (Labelled transition system) *A labeled transition system L is a tuple $(Q, q_0, \Sigma, \rightarrow)$ where:*

- *Q is a set of states,*
- *$q_o \in Q$ is the initial state,*
- *Σ is a set of (visible) symbols called the alphabet, with $\tau \notin \Sigma$ (τ is the internal, invisible or silent action),*
- *$\rightarrow \subseteq Q \times (\Sigma \cup \{\tau\}) \times Q$ is the transition relation.*

♦

Given two states $q, q' \in Q$, for any $a \in (\Sigma \cup \{\tau\})$, we note $q \xrightarrow{a} q'$ if $(q, a, q') \in \rightarrow$, and for any $w = a_1 \cdot a_2 \cdot \ldots \cdot a_n$ in $(\Sigma \cup \{\tau\})^*$, we note $q \xrightarrow{w} q'$

if there exists a sequence of transitions $q_0 \xrightarrow{a_1} q_1 \cdots q_{n-1} \xrightarrow{a_n} q_n$ with $q = q_0$ and $q' = q_n$. Note that if $w = \epsilon$, we have $q = q_0 = q_n = q'$.

Then, given a well formed $_{\mathrm{d}}\mathrm{SL}_\Diamond$ program P and a distribution D of P, we need to define the distributed environment in which P will execute. Indeed, the partition of the variables given by D imposes a partition of the set of `WHEN`s of P. Note that this distributed environment (the distribution of the global variables and the `WHEN`s) has nothing to do with the environment of the control system (i.e. the industrial equipment which is controlled). This distributed environment can be defined as follows.

Definition 5 (Distributed environment) *Given a $_d$SL$_\Diamond$ program P, and a distribution $D = \{V_1, V_2, ..., V_n\}$ of P, we define the distributed environment of P w.r.t. D as follows:*

$$E_D^P = ((V_1, W_1), (V_2, W_2), ..., (V_n, W_n))$$

where each $i \in [1..n], W_i = \{w \in Whens(P) | Var(w) \subseteq V_i\}$. In the following, we call each (V_i, W_i) a local environment and denote it by $(E_D^P)_i$. Note that, since D is a distribution of P, we have $\bigcup_{i\in[1..n]} W_i = Whens(P)$ and $\forall i, j \in [1..n], (i \neq j) \implies (W_i \cap W_j = \emptyset)$ ♦

Moreover, we need to define some auxiliary functions. We first define the executability of an instruction, next we define two functions which construct lists of instructions corresponding respectively to the input and the output of the variables in a given set V.

Definition 6 (Executability) *Given a $_d$SL$_\Diamond$ program P, and a distribution $D = \{V_1, V_2, ..., V_n\}$ of P, we define $Exec_D(\omega)$ (ω is a list of instructions) as the set of sites on which the first instruction of ω can execute. If this first instruction uses a global variable $x \in V_i$, $Exec_D(\omega) = \{i\}$, otherwise $Exec_D(\omega) = \{1, ..., n\}$.* ♦

Definition 7 (Input sampling, output writing) *Given a set of variables V, we define $Sample(V, <_V)$, respectively $Write(V, <_V)$, as the list of instructions performing the input, respectively output, of the (input, resp. output) variables in V in the total order given by $<_V$, as follows:*

$$\begin{aligned} Sample(V, <_V) &= \texttt{INPUT}(\texttt{v}_1); \texttt{INPUT}(\texttt{v}_2); ...; \texttt{INPUT}(\texttt{v}_\texttt{k}) \\ Write(V, <_V) &= \texttt{OUTPUT}(\texttt{v}_1); \texttt{OUTPUT}(\texttt{v}_2); ...; \texttt{OUTPUT}(\texttt{v}_\texttt{k}) \end{aligned}$$

where $\forall i \in [1..k-1], v_i <_V v_{i+1}$ *and* $\bigcup_{i\in[1..k]}\{v_i\} = V$.
$<_V$ *is the order in which the variables are declared in the program text*
♦

The third function constructs a list of instructions corresponding to the processing of `WHEN`s of a set W.

Definition 8 (Treatment of `WHEN`s) *Given a set of* `WHEN`*s* W*, we define* $Treat(W, <_W)$ *as a list of instructions processing all* `WHEN`*s in* W*, in the order given by* $<_W$*, as follows:*

$$Treat(W, <_W) = \omega_1; \omega_2; ...; \omega_{|W|}$$

where $\forall i \in [1..|W|]$:

$$\omega_i = \texttt{IF } (Cond(w_i) \texttt{ AND NOT } old\_cond_{w_i}) \texttt{ THEN } old\_cond_{w_i} := \top; Body(w_i)$$

$$\texttt{ELSE } old\_cond_{w_i} := Cond(w_i) \texttt{ END\_IF}$$

and where $\forall i \in [1..|W|-1], w_i <_W w_{i+1}$ *and* $\bigcup_i\{w_i\} = W$.
$<_W$ *is the order in which the different* `WHEN`*s are defined in the program text*
♦

3.4.3 Structural operational semantics

The formal structural operational semantics for a well formed $_{\mathrm{d}}\mathrm{SL}_\Diamond$ program P, w.r.t. a distribution D, is given below as a labeled transition system whose visible actions are updates to/from the environment of the I/O variables, $\Sigma = Var^{in} \times \{?\} \times \{\top, \bot, \sharp\} \cup Var^{out} \times \{!\} \times \{\top, \bot, \sharp\}$. $\top$ stands for `TRUE`, $\bot$ for `FALSE` and $\sharp$ for `UNKNOWN`. Let us first define a global state of a $_{\mathrm{d}}\mathrm{SL}_\Diamond$ program.

Definition 9 (Global state of a $_{\mathrm{d}}\mathrm{SL}_\Diamond$ program) *Given a distribution* $D = \{V_1, V_2, ..., V_n\}$ *of well formed* $_d SL_\Diamond$ *program* P*, we define a global state* G *of* P *as follows:*

$$G \equiv ((\omega_1, \nu_1, \phi_1), (\omega_2, \nu_2, \phi_2), \ldots, (\omega_n, \nu_n, \phi_n), \sigma_1, \sigma_2, ..., \sigma_\ell, \mu, \xi)$$

where $\forall i \in [1..n], (\omega_i, \nu_i, \phi_i)$ *is the local state of process* i *with the following components:*

- ω_i *is the workload. It gives the sequence of instructions (including extended instructions) remaining to be executed in the current cycle of process* i.

- $\nu_i : (V_i \cup \widetilde{Var(P)} \cup OldCond(W_i)) \mapsto \{\top, \bot, \sharp\}$ *is a valuation function for:*
  - *the global variables owned by process i.*
  - *the tilded (local) copies of all variables.*
  - *the variables containing the old value for the conditions of each* `WHEN` *owned by process i.*
- *Let $\Sigma_\phi = ((Var(P) \times \{\top, \bot, \sharp\} \times \mathbb{N}) \cup \{\diamond_i | i \in [1..n]\})^*$, then $\phi_i \in \Sigma_\phi^*$ is the receiving communication channel of process i. Each message indicates the update of a variable. Additionally, $\diamond_i$ is used to enforce the end of the message treatment.*

and $\forall i \in [1..\ell]$: σ_i is the workload for the `SEQUENCE` *identified by i. It consists of the sequence of instructions remaining to be executed for it.*
$\mu : VarSeq \mapsto \{\top, \bot, \sharp\}$ is the valuation function for the local variables of all `SEQUENCE`*s.*
$\xi : [1..\ell] \mapsto [1..n]$ indicates on which site a given sequence is running.
We will note $\mathcal{G}_D^P$ the set of global states of a $_d$SL$_\diamond$ program P, given a distribution D. ♦

The valuation functions ν_i in each process are defined as expected, except for the evaluation involving the `UNKNOWN` value (denoted $\sharp$) :

$$\nu_i(e) = \begin{cases} \bot & \text{if e = } \texttt{FALSE} \\ \top & \text{if e = } \texttt{TRUE} \\ \sharp & \text{if e = } \texttt{UNKNOWN} \\ \nu_i(e') \overline{\wedge} \nu_i(e'') & \text{if } e = e' \texttt{ AND } e'' \\ \nu_i(e') \overline{\vee} \nu_i(e'') & \text{if } e = e' \texttt{ OR } e'' \\ \overline{\neg} \nu_i(e') & \text{if } e = \texttt{NOT } e' \end{cases}$$

The operators $\overline{\wedge}, \overline{\vee}, \overline{\neg}$ are defined in figure 3.6.
We also use the notation $\nu[X \mapsto e]$ which returns a valuation which is the same as ν, except for all $x \in X$, which are mapped to e. $\nu[x \mapsto e]$ is a shorthand for $\nu[\{x\} \mapsto e]$.
Note that μ is defined in the same way as ν_i.
In order to evaluate expressions which may involve both variables which are defined by the local valuation ν_i of some process i, and local `SEQUENCE` variables, we define Υ_i, which uses ν_i to evaluate $x \notin VarSeq(P)$ and μ to evaluate $x \in VarSeq(P)$.

The following definition will be used to insert delimiters in the FIFO queues of the processes, in order to model the non instantaneous behavior of the network.

x	y	$x\overline{\vee}y$	$x\overline{\wedge}y$	$\overline{\neg}x$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\top$	$\bot$	$\top$
$\bot$	$\sharp$	$\sharp$	$\bot$	$\top$
$\top$	$\bot$	$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\bot$
$\top$	$\sharp$	$\top$	$\sharp$	$\bot$
$\sharp$	$\bot$	$\sharp$	$\bot$	$\sharp$
$\sharp$	$\top$	$\top$	$\sharp$	$\sharp$
$\sharp$	$\sharp$	$\sharp$	$\sharp$	$\sharp$

Figure 3.6: $_{\mathrm{d}}$SL boolean operators

Definition 10 (FIFO Shuffle) *Given a* FIFO *queue* $\phi \in \Sigma_\phi^*$*, and a permutation* π *of* $[1..n]$*, we define* $Shuffle(\phi)_\pi = \{\phi_1 \cdot \diamond_{\pi(1)} \cdot \phi_2 \cdot \diamond_{\pi(2)} \cdots \phi_n \cdot \diamond_{\pi(n)} \cdot \phi_{n+1} | \forall i \in [1..n] : \phi_i \in \Sigma_\phi^* \wedge \phi_1 \cdot \phi_2 \cdots \phi_{n+1} = \phi\}$. ♦

To keep the semantic rules compact, we introduce only one FIFO for each process. However, as we shall see, each FIFO is modeled in such a way that it contains sub-channels for each originating site. The following definition extracts one subchannel from a given FIFO queue.

Definition 11 (FIFO Subchannel) *Let* $\Sigma_{\phi_i} = \{(x, v, s) | x \in Var(P), v \in \{\top, \bot, \sharp\}, s = i\}$*. Given a* FIFO *queue* $\phi \in \Sigma_\phi^*$*,* $SubCh_i(\phi)$ *is the projection of* ϕ *on* Σ_{ϕ_i}*, trimming* ϕ *at* $\diamond_i$*, i.e.,* $SubCh_i(\phi)$ *is defined recursively as follows :*

1. $SubCh_i(\epsilon) = \epsilon$
2. $SubCh_i(m \cdot \phi') = \begin{cases} m \cdot SubCh_i(\phi') & \text{if } m \in \Sigma_{\phi_i} \\ SubCh_i(\phi') & \text{if } m \notin \Sigma_{\phi_i} \wedge m \neq \diamond_i \\ \epsilon & \text{if } m = \diamond_i \end{cases}$
   with $m \in \Sigma_\phi$ *and* $\phi' \in \Sigma_\phi^*$

♦

We also need to extract the first message from a given site from the FIFO channel. The following definition extracts the first message, if any, from the channel.

Definition 12 (Message removal) *Given a* FIFO *queue* $\phi \in \Sigma_\phi^*$*,* $i \in [1..n]$*.*

1. *If* $SubCh_i(\phi) = \epsilon$*, then* $RemoveMsg_i(\phi) = \phi$*.*

2. *Otherwise,* $RemoveMsg_i(\phi) = \phi' \cdot \phi''$ *if* $\phi = \phi' \cdot (x, v, i) \cdot \phi''$ *with* $SubCh_i(\phi') = \epsilon$

♦

The last operation on FIFO queues removes the delimiters introduced by $Shuffle(\phi)_\pi$.

Definition 13 (Delimitor removal) *Given* $\phi \in \Sigma^*_\phi$. $RemoveMark(\phi)$ *is the projection of* ϕ *on* $\Sigma^*_\phi \setminus \{\diamond_i | i \in [1..n]\}$. ♦

We can now introduce the semantic rules which will provide the transition relation of the labeled transition system. The first two rules are global rules acting on the entire global state. The first one expresses the interleaving semantics; i.e., if in a local state a transition can be taken, then from any global state containing this local state, the same transition can be taken, only modifying that local state. It can be noticed here that, at a global level, and contrary to synchronous languages like Esterel or Lustre, $_{\text{d}}\text{SL}_\diamond$ has an asynchronous semantics

[Interleaving]

$$\frac{(E^P_D)_i \vdash (\omega_i, \nu_i, \phi_i) \xrightarrow{a} (\omega'_i, \nu'_i, \phi'_i)}{E^P_D \vdash ((\omega_1, \nu_1, \phi_1), ..., (\omega_i, \nu_i, \phi_i), ..., (\omega_n, \nu_n, \phi_n), \sigma_1, ..., \sigma_\ell, \mu, \xi) \xrightarrow{a} ((\omega_1, \nu_1, \phi_1), ..., (\omega'_i, \nu'_i, \phi'_i), ..., (\omega_n, \nu_n, \phi_n), \sigma_1, ..., \sigma_\ell, \mu, \xi)} \quad \forall a \in \{\tau\} \cup \Sigma$$

The second global rule corresponds to the broadcast. If a process has to perform a broadcast corresponding to a change of value of a variable, then a τ-transition is taken, leading to a global state where all the receiving communication channels are updated with a message. Note that the message is also put in the channel of the process performing the broadcast. Indeed, this local process might have an asynchronous (tilded) copy of its own variable, and this copy needs to be (asynchronously) updated as well.

[Broadcast]

$$E^P_D \vdash ((\omega_1, \nu_1, \phi_1), ..., (\texttt{BCAST(x)};\omega_i, \nu_i, \phi_i), ..., (\omega_n, \nu_n, \phi_n), \sigma_1, ..., \sigma_\ell, \mu, \xi) \xrightarrow{\tau} ((\omega_1, \nu_1, \phi'_1), ..., (\omega_i, \nu_i, \phi'_i), ..., (\omega_n, \nu_n, \phi'_n), \sigma_1, ..., \sigma_\ell, \mu, \xi) \quad \text{where } \forall j \in [1..n] : \phi'_j = \phi_j \cdot (x, \nu_i(x), i)$$

The next set of rules are local rules; i.e. defining how a local process makes a move. The first of these local rules corresponds to the beginning of a new

cycle. If, at a point in the execution, the workload of process i becomes empty (ε), then a new *Input-Process-Output* cycle is scheduled in the workload. Note that all samples are taken before events are triggered. This is consistent with the idea of *instants*, where a snapshot is taken from the environment, and reactions are based upon this snapshot. This rule dictates the cyclic behavior of each process. Note that, in order to model the non instantaneous behavior of the network, the $\diamond_i$ markers are inserted to delimit the messages that will be treated during this cycle. The markers also model the fact that messages which are arriving during the current cycle will not be treated (cfr message treatment rule further on).

[Cycle start]

$$(E_D^P)_i \vdash (\varepsilon, \nu_i, \phi_i) \xrightarrow{\tau} (Sample(Var^{in}(P) \cap V_i, <_V); Treat(W_i, <_W); \texttt{MSG}; Write(Var^{out}(P) \cap V_i, <_V), \nu_i, \phi_i')$$

$$\phi_i' \in Shuffle(\phi_i)_\pi \text{ for some } \pi$$

The second local rule describes the sampling of an input from the environment. When doing so, the valuation needs to be updated according to this sampling, and the new value needs to be broadcast to all processes. Moreover, the transition is labeled with the sampled variable and the value that has been read.

[Input]

$$(E_D^P)_i \vdash (\texttt{INPUT(x)};\omega_i, \nu_i, \phi_i) \xrightarrow{x?a} (\texttt{BCAST(x)};\omega_i, \nu_i[x \mapsto a], \phi_i)$$

$$\forall a \in \{\top, \bot\}$$

The following two rules describe the message treatment phase. In this phase, some messages are read from the receiving channel and the local valuation is updated accordingly (i.e. the local asynchronous - tilded - copy is set to the new value). Due to this change of valuation, some `WHEN`'s might be triggered. Thus, all `WHEN`'s that might be triggered by this change of valuation are considered before continuing the message treatment. Note that messages are of the form (x, v, s) where x is the updated variable, v is its value, and s is the originating process. The first message from any sender may be received by the process. This is a technical subtlety to make the FIFO queue of one process behave as if there was a FIFO queue for each process.

[Message treatment]

$$(E_D^P)_i \vdash (\texttt{MSG};\omega_i, \nu_i, \phi_i) \xrightarrow{\tau} (Treat(W_{i/\tilde{x}}, <_W); \texttt{MSG};\omega_i, \nu_i[\widetilde{x} \mapsto v], RemoveMsg_j(\phi_i))$$

For any $j \in [1..n]$ such that $SubCh_j(\phi_i) = (x, v, s) \cdot \phi'$

where $W_{i/\widetilde{x}} = \{w \in W_i \mid \widetilde{x} \in Var(Cond(w))\}$

[End of message treatment]

$$(E_D^P)_i \vdash (\texttt{MSG};\omega_i, \nu_i, \phi_i) \xrightarrow{\tau} (\omega_i, \nu_i, RemoveMark(\phi_i))$$

$$\text{if } \forall j \in [1..n] : SubCh_j(\phi_i) = \epsilon$$

Note that the reception may not be instantaneous, that is, the end of message treatment rule may be applied while there are still some messages left in the FIFO channel. The $\diamond_i$ markers, inserted when the cycle start rule was fired, assures that new messages received during this cycle are not treated.

The next rule corresponds to the treatment of an assignment in the workload. The assignment has the usual effect (the local valuation is updated). However, if the assigned variable is not an $old\_cond_{w_i}$, the new value needs to be broadcast and the `WHEN`s that may be triggered by this assignment need to be scheduled for treatment. Notice that the value of x is sent to all sites, regardless of the use of $\sim x$ on the destination sites. This is of course not necessary, and the actual implementation does not.

[Assignment]

$$(E_D^P)_i \vdash (x := e; \omega_i, \nu_i, \phi_i) \xrightarrow{\tau} \begin{cases} (\texttt{BCAST(x)};Treat(W_{i/x}, <_W); \omega_i, \nu_i[x \mapsto \nu_i(e)], \phi_i) & \\ \qquad\qquad \text{if } x \in Var(P) & \\ (\omega_i, \nu_i[x \mapsto \nu_i(e)], \phi_i) \qquad \text{if } x \in OldCond(P) & \end{cases}$$

The following rule corresponds to the treatment of an `IF` statement. As expected, if the condition evaluates to $\top$, the code of the `THEN` part is inserted in the workload, otherwise (the condition evaluated to $\bot$), the code of the `ELSE` part is inserted. The statement is skipped if the condition is `UNKNOWN`.

[If]

$$(E_D^P)_i \vdash (\texttt{IF}\, e\, \texttt{THEN}\, \omega_\top\, \texttt{ELSE}\, \omega_\bot\, \texttt{ENDIF};\omega_i, \nu_i, \phi_i) \xrightarrow{\tau} \begin{cases} (\omega_\top;\omega_i, \nu_i, \phi_i) & \text{if } \nu_i(e) = \top \\ (\omega_\bot;\omega_i, \nu_i, \phi_i) & \text{if } \nu_i(e) = \bot \\ (\omega_i, \nu_i, \phi_i) & \text{if } \nu_i(e) = \sharp \end{cases}$$

The last local rule corresponds to the output of a variable. In this case, nothing needs to be done, apart from firing a transition labeled by the output variable and its new value, and removing this abstract instruction from the workload.

[Output]

$$(E_D^P)_i \vdash (\texttt{OUTPUT(x)};\omega_i, \nu_i, \phi_i) \xrightarrow{x!\nu_i(x)} (\omega_i, \nu_i, \phi_i)$$

The last set of rules are global rules, describing how SEQUENCEs are handled. Remember that SEQUENCEs can execute in a completely distributed manner, which is reflected here by the global nature of the rules.

In the first two rules, the activation of a SEQUENCE is formalized. In the first rule, the SEQUENCE is started from the local workload ω_i of process i, by the instruction LAUNCH s. This instruction is removed from the workload, and the corresponding SEQUENCE's body is placed into σ_s. The local variables are set to UNKNOWN, by updating μ. Finally, the SEQUENCE is started on any site that can execute the first instruction, by changing ξ accordingly. Note that this rule can only be fired when there is no running instance of the SEQUENCE. The second rule states that if an instance is already running ($\sigma_s \neq \epsilon$), the LAUNCH instruction is simply removed from the local workload.

[Sequence Activate]

$$E_D^P \vdash ((\omega_1,\nu_1,\phi_1),...,(\omega_i,\nu_i,\phi_i),...,(\omega_n,\nu_n,\phi_n),\sigma_1,...,\sigma_s,...,\sigma_\ell,\mu,\xi)$$
$$\xrightarrow{\tau} ((\omega_1,\nu_1,\phi_1),...,(\omega_i',\nu_i,\phi_i),...,(\omega_n,\nu_n,\phi_n),\sigma_1,...,\sigma_s',...,\sigma_\ell,\mu',\xi')$$
$$\text{for any } i \in [1..n], s \in [1..\ell], \text{ such that } \sigma_s = \epsilon$$

$\omega_i = \texttt{LAUNCH}\ s\ \texttt{;}\ \omega_i'$
$\sigma_s' = \mathcal{S}(s)$
$\xi' = \xi[s \mapsto j]$ for any $j \in Exec_D(\mathcal{S}(s))$
$\mu' = \mu[VarSeq(s) \mapsto \sharp]$

[Sequence Activate']

$$E_D^P \vdash ((\omega_1,\nu_1,\phi_1),...,(\omega_i,\nu_i,\phi_i),...,(\omega_n,\nu_n,\phi_n),\sigma_1,...,\sigma_s,...,\sigma_\ell,\mu,\xi)$$
$$\xrightarrow{\tau} ((\omega_1,\nu_1,\phi_1),...,(\omega_i',\nu_i,\phi_i),...,(\omega_n,\nu_n,\phi_n),\sigma_1,...,\sigma_s,...,\sigma_\ell,\mu,\xi)$$
$$\text{for any } i \in [1..n], s \in [1..\ell], \text{ such that } \sigma_s \neq \epsilon$$

where $\omega_i = \texttt{LAUNCH}\ s\ \texttt{;}\ \omega_i'$

The following rule describes how an assignment is handled inside a SEQUENCE. A SEQUENCE will only be executed during the message treatment phase, hence the abstract instruction MSG must be in front of the local workload ω_i. When an assignment instruction is executed in a SEQUENCE s, the local valuation ν_i is updated, if it updates a global variable. On the other hand, if the left hand side consists of a local variable, the valuation μ is updated. As usual, the instruction is removed from the instructions remaining to be executed for this SEQUENCE, and WHENs that may be triggered by the

assignment are scheduled for treatment in the local workload ω_i. Finally, since SEQUENCEs can execute in a distributed manner, they may change from site to site, which is expressed here by the update of ξ, which *moves* the SEQUENCE to any site capable of executing its first (new) instruction.

[Sequence Assign]

$$E_D^P \vdash ((\omega_1,\nu_1,\phi_1),...,(\omega_i,\nu_i,\phi_i),...,(\omega_n,\nu_n,\phi_n),\sigma_1,...,\sigma_s,...,\sigma_\ell,\mu,\xi)$$
$$\xrightarrow{\tau} ((\omega_1,\nu_1,\phi_1),...,(\omega_i',\nu_i',\phi_i),...,(\omega_n,\nu_n,\phi_n),\sigma_1,...,\sigma_s',...,\sigma_\ell,\mu',\xi')$$
$$\text{for any } i \in [1..n], s \in [1..l], \text{ such that } \xi(s) = i$$

$\omega_i = \texttt{MSG;}\ \omega_i''$
$\sigma_s = \texttt{x := e;}\ \sigma_s'$
$\nu_i' = \nu_i[x \mapsto \Upsilon_i(e)]$ if $x \notin VarSeq$, otherwise $\nu_i' = \nu_i$
$\mu' = \mu_i[x \mapsto \Upsilon_i(e)]$ if $x \in VarSeq$, otherwise $\mu' = \mu$
$\omega_i' = \texttt{BCAST(x)};Treat(W_{i/x}, <_W);\omega_i$ if $x \notin VarSeq$, otherwise $\omega_i' = \omega_i$
$\xi' = \xi[s \mapsto j]$ for any $j \in Exec(\sigma_s')$

The next rule describes how an IF instruction inside a SEQUENCE is handled. When the process on which the SEQUENCE currently executes is in its message treatment phase, it replaces the IF instruction by the correct branch, depending on the evaluation of the IF's condition e. The SEQUENCE may change sites, which is modeled by the update of ξ.

[Sequence IF]

$$E_D^P \vdash ((\omega_1,\nu_1,\phi_1),...,(\omega_i,\nu_i,\phi_i),...,(\omega_n,\nu_n,\phi_n),\sigma_1,...,\sigma_s,...,\sigma_\ell,\mu,\xi)$$
$$\xrightarrow{\tau} ((\omega_1,\nu_1,\phi_1),...,(\omega_i,\nu_i,\phi_i),...,(\omega_n,\nu_n,\phi_n),\sigma_1,...,\sigma_s',...,\sigma_\ell,\mu,\xi')$$
$$\text{for any } i \in [1..n], s \in [1..l], \text{ such that } \xi(s) = i$$

$\omega_i = \texttt{MSG;}\ \omega_i'$
$\sigma_s = \texttt{IF (e) THEN}\ \sigma_\top\ \texttt{ELSE}\ \sigma_\bot\ \texttt{ENDIF;}\ \sigma_s''$
$\sigma_s' = \sigma_\top\ \texttt{;}\ \sigma_s''$ if $\Upsilon_i(e) = \top$
$\sigma_s' = \sigma_\bot\ \texttt{;}\ \sigma_s''$ if $\Upsilon_i(e) = \bot$
$\sigma_s' = \sigma_s''$ if $\Upsilon_i(e) = \sharp$
$\xi' = \xi[s \mapsto j]$ for any $j \in Exec(\sigma_s')$

The last rule describes how a WHILE instruction inside a SEQUENCE is transformed into an IF instruction. Notice that this is a pure syntactical transformation, and that μ, ξ, and all local components of process i remain unchanged.

[Sequence WHILE]

$$E_D^P \vdash ((\omega_1, \nu_1, \phi_1), ..., (\omega_i, \nu_i, \phi_i), ..., (\omega_n, \nu_n, \phi_n), \sigma_1, ..., \sigma_s, ..., \sigma_\ell, \mu, \xi)$$
$$\xrightarrow{\tau} ((\omega_1, \nu_1, \phi_1), ..., (\omega_i, \nu_i, \phi_i), ..., (\omega_n, \nu_n, \phi_n), \sigma_1, ..., \sigma'_s, ..., \sigma_\ell, \mu, \xi)$$
$$\text{for any } i \in [1..n], s \in [1..l], \text{ such that } \xi(s) = i$$

$\omega_i = \texttt{MSG;}\ \omega'_i$
$\sigma_s = \texttt{WHILE (e) DO}\ \sigma_w\ \texttt{END\_WHILE}\ \sigma''_s$
$\sigma'_s = \texttt{IF (e) THEN}\ \sigma_w\ \texttt{;}\ \texttt{WHILE (e) DO}\ \sigma_w\ \texttt{END\_WHILE; END\_IF}\ \sigma''_s$

Using those semantic rules, given a well-formed $_{\text{d}}\text{SL}_\diamondsuit$ program P and a distribution D of P, we can define a labeled transition system describing the behavior of P, under D.

Definition 14 (Distributed semantics of a $_{\text{d}}\text{SL}_\diamondsuit$ program) *Given a well-formed $_d$SL$_\diamondsuit$ program P and a distribution $D = \{V_1, ..., V_n\}$ of P, we define the distributed semantics of P w.r.t. D, noted $[\![P]\!]_D$ as the labeled transition system $(\mathcal{G}_D^P, G_D^0, (V(P) \times \{!, ?\} \times \{\top, \bot\}), \rightarrow)$*

- $G_D^0 = ((\omega_1, \nu_1, \phi_1), ..., (\omega_n, \nu_n, \phi_n), \sigma_1, ..., \sigma_\ell, \mu, \xi)$
  where $\forall i \in [1..n]$:
  - $\omega_i = \varepsilon$
  - $\nu_i(x) = \sharp$, $\forall x \in (V_i \cup \widetilde{Var(P)} \cup OldCond(W_i))$
  - $\phi_i = \varepsilon$

  $\forall i \in [1..l] : \sigma_i = \epsilon \wedge \xi(i) = 1$
  $\mu(x) = \sharp, \forall x \in VarSeq$

- $\rightarrow$ *is such that* $(G, a, G') \in \rightarrow$ *if and only if* $E_D^P \vdash G \xrightarrow{a} G'$ *can be derived from any structural operational semantic rule given previously.*

♦

3.5 Properties of $_{\text{d}}$SL's semantics

3.5.1 A lattice of behaviors

For a given program P, we have different possible labeled transition systems, one for each possible distribution of P. We can compare these structures to have an idea of how the distribution influences the behavior of P.

Definition 15 (Weak simulation relation) *Given two labeled transition systems* $L_1 = (Q_1, q_1^0, \Sigma, \rightarrow_1)$, $L_2 = (Q_2, q_2^0, \Sigma, \rightarrow_2)$. *A binary relation* $R \subseteq Q_1 \times Q_2$ *is a weak simulation relation for* L_1 *and* L_2 *if and only if for all* $q_1 \in Q_1, q_2 \in Q_2, a \in \Sigma \cup \{\tau\}$, *if* $(q_1, q_2) \in \mathcal{R}$ *then*

$$\forall q_1' : \left(q_1 \xrightarrow{a} q_1'\right) \implies \left(\exists q_2' : q_2 \xrightarrow{\widehat{a}} q_2' \wedge (q_1', q_2') \in \mathcal{R}\right)$$

where

$$\widehat{a} \in \begin{cases} \tau^* \cdot a \cdot \tau^* & \text{if } a \in \Sigma \\ \tau^* & \text{if } a \notin \Sigma \text{ (i.e. } a = \tau\text{)} \end{cases}$$

♦

A LTS $L_1 = (Q_1, q_1^0, \Sigma, \rightarrow_1)$ can be simulated by a LTS $L_2 = (Q_2, q_2^0, \Sigma, \rightarrow_2)$ noted $L_1 \lesssim L_2$ if there exists a weak simulation relation $\mathcal{R}$ for L_1 and L_2 such that $(q_1^0, q_2^0) \in \mathcal{R}$.

In the previous section, we have defined the behavior of a program P with distribution D as a labeled transition system. In this section, we use this structure to prove that if a distribution D' of P refines distribution D of P, then $[\![P]\!]_D$ can be simulated by $[\![P]\!]_{D'}$, and therefore has less behaviors. For that, we need some intermediate results.

Definition 16 (Workload distribution) *Let* $\omega, \omega_1, \omega_2$ *be three workloads (i.e. sequences of instructions, including abstract instructions). We have that* (ω_1, ω_2) *is a distribution of* ω, *noted* $(\omega_1, \omega_2) \succ \omega$, *if and only if*

$\omega = \epsilon \wedge \omega_1 = \epsilon \wedge \omega_2 = \epsilon$
or
$\omega = x;\omega'$ *and one of the following holds:*
$x = \texttt{MSG} \wedge \omega_1 = \texttt{MSG};\omega_1' \wedge \omega_2 = \texttt{MSG};\omega_2' \wedge (\omega_1', \omega_2') \succ \omega'$
$x \notin \{\texttt{MSG}, \epsilon\} \wedge \omega_1 = x;\omega_1' \wedge (\omega_1', \omega_2) \succ \omega'$
$x \notin \{\texttt{MSG}, \epsilon\} \wedge \omega_2 = x;\omega_2' \wedge (\omega_1, \omega_2') \succ \omega'$

♦

The workload distribution will be used to prove the simulation relation between $[\![P]\!]_D$ and $[\![P]\!]_{D'}$, where D' is equal to D, except for one site, which is split into two sites. ω represents the workload of the process which is split into ω_1 and ω_2 in distribution D'. The workload distribution states that the workloads are related when they are empty, or when an instruction is on workload ω, it should be on either ω_1 or ω_2, unless it is a `MSG` instruction, which must then be on both workloads.

Definition 17 (FIFO distribution) *Let ϕ, ϕ_1, ϕ_2 be three FIFOs. We have that (ϕ_1, ϕ_2) is a distribution of ϕ, noted $(\phi_1, \phi_2) \approx_n \phi$, if and only if one of the following holds:*

1. $\phi = \epsilon \wedge \phi_1 = \epsilon \wedge \phi_2 = \epsilon$
2. $\phi = (x, v, s) \cdot \phi' \wedge \phi_1 = (x_1, v_1, s_1) \cdot \phi'_1$
   $\wedge \phi_2 = (x_2, v_2, s_2) \cdot \phi'_2 \wedge (\phi'_1, \phi'_2) \approx_n \phi'$
   with
   $s \neq n \rightarrow (s_1 = s \wedge s_2 = s)$
   $s = n \rightarrow (s_1 = s_2 = n \vee s_1 = s_2 = n + 1)$
3. $\phi = \diamond_i \cdot \phi'$
   $\wedge (i \neq n \rightarrow (\phi_1 = \diamond_i \cdot \phi'_1 \wedge \phi_2 = \diamond_i \cdot \phi'_2 \wedge (\phi'_1, \phi'_2) \approx_n \phi'))$
   $\wedge (i = n \rightarrow (\phi_1 = \diamond_n \cdot \diamond_{n+1} \cdot \phi'_1 \wedge \phi_2 = \diamond_n \cdot \diamond_{n+1} \cdot \phi'_2 \wedge (\phi'_1, \phi'_2) \approx_n \phi'))$

♦

The FIFO distribution will be used in a similar way as the workload distribution. It states that FIFO queues ϕ, ϕ_1 and ϕ_2 are related if they are the same, or if there is a message on ϕ which came from process n, it should be present on ϕ_1 and ϕ_2, but with possibly a different originating process $(n + 1)$. The use of n and $n + 1$ results from the simplification in proof 1, where the last process in D is split to form D'.

Lemma 1 (One-split simulation) *Given a well-formed $_d$SL$_\diamond$ program P and two distributions $D = (V_1, ..., V_n)$, $D' = (V'_1, ..., V'_n, V'_{n+1})$ of P such that $\forall i \in [1..n-1], V_i = V'_i$ and $V_n = V'_n \cup V'_{n+1}$. We have that $[\![P]\!]_{D'}$ simulates $[\![P]\!]_D$:*

$$[\![P]\!]_D \lesssim [\![P]\!]_{D'}$$

Proof sketch. (Complete proof is given in section 3.5.2) We define a relation $\mathcal{R} \subseteq \mathcal{G}^P_D \times \mathcal{G}^P_{D'}$ such that
if $G = ((\omega^G_1, \nu^G_1, \phi^G_1), (\omega^G_2, \nu^G_2, \phi^G_2), ..., (\omega^G_n, \nu^G_n, \phi^G_n), \sigma^G_1, \sigma^G_2, ..., \sigma^G_\ell, \mu^G, \xi^G)$
and
$G' = ((\omega^{G'}_1, \nu^{G'}_1, \phi^{G'}_1), (\omega^{G'}_2, \nu^{G'}_2, \phi^{G'}_2), ..., (\omega^{G'}_n, \nu^{G'}_n, \phi^{G'}_n), (\omega^{G'}_{n+1}, \nu^{G'}_{n+1}, \phi^{G'}_{n+1}),$
$\sigma^{G'}_1, \sigma^{G'}_2, ..., \sigma^{G'}_\ell, \mu^{G'}, \xi^{G'})$, $(G, G') \in \mathcal{R}$ if and only if:

1. $(\omega^G_i, \nu^G_i, \phi^G_i) = (\omega^{G'}_i, \nu^{G'}_i, \phi^{G'}_i)$, $\forall i \in [1..n-1]$
2. $(\phi^{G'}_n, \phi^{G'}_{n+1}) \approx_n \phi^G_n$
3. $\forall x \in (V'_n \cup \widetilde{Var(P)} \cup OldCond(W'_{n+1}))$, $\nu^G_n(x) = \nu^{G'}_n(x)$

4. $\forall x \in (V'_{n+1} \cup \widetilde{Var(P)} \cup OldCond(W'_n)),\ \nu_n^G(x) = \nu_{n+1}^{G'}(x)$
5. $(\omega_n^{G'}, \omega_{n+1}^{G'}) \succ \omega_n^G$
6. $\forall i \in [1..\ell] : \sigma_i^G = \sigma_i^{G'}$
7. $\mu^G = \mu^{G'}$
8. $\forall i \in [1..\ell] : \begin{cases} \xi^G(i) < n & \rightarrow \xi^G(i) = \xi^{G'}(i) \\ \xi^G(i) = n & \rightarrow \xi^{G'}(i) \in \{n, n+1\} \end{cases}$

Then we prove that $(G_0^D, G_0^{D'}) \in \mathcal{R}$ *and that* $\mathcal{R}$ *is a weak simulation relation for* $[\![P]\!]_D$ *and* $[\![P]\!]_{D'}$*, i.e. we prove that if* $(G, G') \in \mathcal{R}$ *then, for all* $a \in \Sigma \cup \{\tau\}$*:*

$$\forall H\ ,\ \left(G \xrightarrow{a} H\right) \implies \left(\exists H'\ ,\ G' \xrightarrow{\widehat{a}} H' \wedge ((H, H') \in \mathcal{R})\right)$$

We must prove that every possible step made from G *to* H *can be simulated from* G' *to* H'*. In short, the workload distribution assures that all instructions executed by the processes in* G *can also be executed in* G'*. If process* n *executes some instruction in* G*, then either process* n *or process* $n+1$ *must be able to do so in* G'*. The* FIFO *distribution assures that sent messages in* G *are also present in* G'*. Messages sent by process* n *in* G *are also sent in* G'*, either by process* n *or process* $n+1$*. A similar reasoning governs* ξ*, where a* `SEQUENCE` *running on process* n *in* G *must either be running on process* n *or* $n+1$ *in* G'*.*

To prove this in detail, we need to consider every possible step described in the semantics. ■

Lemma 1 proves that, having two distributions D and D' whose difference is that the last set of variables of D is split in two in D', $[\![P]\!]_D \lesssim [\![P]\!]_{D'}$. A full proof of lemma 1 can be found in section 3.5.2. We now prove for any distribution D, D' such that $D \preceq D'$, $[\![P]\!]_D \lesssim [\![P]\!]_{D'}$ holds. For this, we first give a lemma that will allow us to relate this problem to lemma 1.

Lemma 2 (Distribution decomposition) *Let* $D = \{V_1, V_2, ..., V_k\}$ *and* $D' = \{V'_1, V'_2, ..., V'_l\}$ *be two distributions such that* $D \preceq D'$*. We have that there exists a finite sequence of distributions* $D_1, ..., D_n$ *(note that* $n = l - k + 1$*), such that*

$$D = D_1 \preceq D_2 \preceq ... \preceq D_n = D'$$

such that $\forall i \in [2..n] : D_i$ *is obtained by one split of* D_{i-1} ■

Proof. Note first that $l > k$. Indeed, if $l < k$, clearly $D \not\preceq D'$, if $l = k$ and $D \preceq D'$, clearly $D = D'$.
Since $l > k \wedge D \preceq D'$, there must be $p, q \in [1..l]$ such that $V'_p \subsetneq V_i \wedge V'_q \subsetneq V_i$ for some $i \in [1..k]$ (Pigeonhole principle). For simplicity we suppose $p = l, q = l - 1$. This implies that $V'_l \cup V'_{l-1} \subseteq V_i$. Hence the distribution $D'' = \{V'_1, V'_2, ..., V'_{l-2}, V_{l-1} \cup V_l\}$ still refines D, but has one element less than D'. This process can be repeated until $D'' = D$, and consists of only one split refinements. □

Remark *Note that the order of the elements in D is of no importance in lemma 2. For notational simplicity, we suppose that it is the last partition that is split.*

With this lemma, we know that if $D \preceq D'$, then we can construct a sequence of distributions and we can apply lemma 1 between each consecutive pair of distributions of this sequence.

Theorem 2 (Simulation) *Given a well-formed $_d$SL$_\diamond$ program $P = (V, \prec_V, W, \prec_W)$, let $D = \{V_1, V_2, ..., V_n\}$ and $D' = \{V'_1, V'_2, ..., V'_l\}$ be two distributions of P. We have that, if D' refines D then $[\![P]\!]_{D'}$ simulates $[\![P]\!]_D$:*

$$(D \preceq D') \implies ([\![P]\!]_D \lesssim [\![P]\!]_{D'})$$

This is a direct consequence of lemma 1, lemma 2, and the transitivity of the simulation relation. ■

Intuitively, this means that every step that P can perform with distribution D can be simulated by P using a more refined distribution D'.

Corollary 1 (Distribution lattice) *Given a well-formed $_d$SL$_\diamond$ program. Let $D_{min} = \{Var(P)\}$. We have that $\langle \{[\![P]\!]_D | D \in \mathcal{D}_P\}, \lesssim \rangle$ is a lattice with, respectively, $[\![P]\!]_{D_{min}}$ and $[\![P]\!]_{D_{max}}$ as minimal and maximal elements.* ■

Corollary 2 (Property preservation) *Given a $_d$SL$_\diamond$ program P, and two distributions D, D' such that $D \preceq D'$ we have for all next-free* LTL *formula ϕ :*

$$[\![P]\!]_{D'} \models \phi \Rightarrow [\![P]\!]_D \models \phi$$

■

Therefore, a program P is safe (there is no run which violates a certain property) for any distribution D if it is safe for the maximal distribution D_{max}. On the other hand, if P is shown not safe for some distribution D it is also not safe for any more refined distribution D'.

3.5.2 Full proof of the one-split simulation

Lemma 3 *Given a well-formed $_d$SL$_\diamond$ program P and two distributions of P, $D = (V_1, ..., V_n)$, $D' = (V'_1, ..., V'_n, V'_{n+1})$ such that $\forall i \in [1..n-1] : V_i = V'_i$ and $V_n = V'_n \cup V'_{n+1}$, let $(E^P_D)_n = (V_n, W_n)$, $(E^P_{D'})_n = (V'_n, W'_n)$ and $(E^P_{D'})_{n+1} = (V'_{n+1}, W'_{n+1})$, we have :*

$$\begin{aligned} Var^{in}(P) \cap V_n &= (Var^{in}(P) \cap V'_n) \cup (Var^{in}(P) \cap V'_{n+1}) \\ Var^{\tau}(P) \cap V_n &= (Var^{\tau}(P) \cap V'_n) \cup (Var^{\tau}(P) \cap V'_{n+1}) \\ Var^{out}(P) \cap V_n &= (Var^{out}(P) \cap V'_n) \cup (Var^{out}(P) \cap V'_{n+1}) \\ W_n &= W'_n \cup W'_{n+1} \end{aligned}$$

■

Proof. Results directly from the definition of distribution, the nature of D, D' and the definition of $Var^{in}(P), Var^{\tau}(P), Var^{out}(P)$. □

This lemma states that when the last process is split in D, the input, output and internal variables together with the `WHEN`s are split accordingly.

Lemma 4 (Distributed executability) *Given a well-formed $_d$SL$_\diamond$ program P and two distributions $D = (V_1, ..., V_n)$, $D' = (V'_1, ..., V'_n, V'_{n+1})$ of P such that $\forall i \in [1..n-1], V_i = V'_i$ and $V_n = V'_n \cup V'_{n+1}$. We have that $\forall \omega$:*
$Exec_D(\omega) = \{n\} \Rightarrow Exec_{D'}(\omega) \in \{n, n+1\}$
and $Exec_D(\omega) = \{i\} \wedge i < n \Rightarrow Exec_{D'}(\omega) = \{i\}$ ■

Proof. This results directly from the definition of $Exec_D$ and the nature of D and D'. □

The following lemma states that if three FIFO queues ϕ, ϕ_1 and ϕ_2 are related by $\approx_n$, and a message is in some subchannel $i < n$ of ϕ, then the same message can also be found in the same subchannel of ϕ_1 and ϕ_2. For messages in subchannel n of ϕ, they are present either in subchannel n or in subchannel $n+1$ of ϕ_1 and ϕ_2.

Lemma 5 *For any three FIFO channels $\phi, \phi_1, \phi_2 \in \Sigma^*_\phi$, we have that if $\phi \approx_n (\phi_1, \phi_2)$, then $\forall s \in [1..n-1] : SubCh_s(\phi) = SubCh_s(\phi_1) = SubCh_s(\phi_2)$. Moreover, if $SubCh_n(\phi) = (x, v, n) \cdot \phi'$, then $SubCh_n(\phi_1) = SubCh_n(\phi_2) = (x, v, n) \cdot \phi''$ or $SubCh_{n+1}(\phi_1) = SubCh_{n+1}(\phi_2) = (x, v, n+1) \cdot \phi'''$.* ■

Proof. We will prove this property by induction on the size of ϕ. Suppose $\phi = \epsilon$, then $\phi_1 = \phi_2 = \epsilon$, and the property holds. Now suppose that $\phi = m \cdot \phi'$ and that the lemma holds for ϕ' w.r.t. ϕ'_1 and ϕ'_2. Either m is a message (x, v, s), or $m \in \{\diamond_i | i \in [1..n]\}$.

1. $m = (x, v, s)$. In this case, since $\phi \approx_n (\phi_1, \phi_2)$ we must have that $\phi_1 = (x, v, s') \cdot \phi'_1$ and $\phi_2 = (x, v, s') \cdot \phi'_2$. If $s \neq n$ then $s' = s$ by definition of $\approx_n$. It is easy to see by definition of $SubCh_i$ that this only modifies $SubCh_s(\phi)$, and that the lemma still holds. On the other hand, if $m = (x, v, n)$, we must either have (x, v, n) or $(x, v, n+1)$ in front of ϕ_1 and ϕ_2 to maintain $\phi \approx_n (\phi_1, \phi_2)$. If (x, v, n) is inserted in ϕ_1 and ϕ_2, this will only change $SubCh_n$. From the definition of $SubCh_n$, we can see that the lemma still holds. For $(x, v, n+1)$, the same reasoning holds.

2. $m = \diamond_i$. In this case, for $\phi \approx_n (\phi_1, \phi_2)$ to hold, either $i \neq n$ and we must have $\phi_1 = \diamond_i \cdot \phi'_1$, $\phi_2 = \diamond_i \cdot \phi'_2$. By definition of $SubCh_i$, this does not alter any subchannels, except for subchannel i which is truncated to ϵ. So the lemma holds for $i \in [1..n-1]$. If $i = n$, $\phi_1 = \diamond_n \cdot \diamond_{n+1} \cdot \phi'_1$, $\phi_2 = \diamond_n \cdot \diamond_{n+1} \cdot \phi'_2$. By definition of $SubCh_i$, remark that this truncates subchannels n and $n+1$. We have therefore $SubCh_n(\phi) = SubCh_n(\phi_1) = SubCh_n(\phi_2) = SubCh_{n+1}(\phi_1) = SubCh_{n+1}(\phi_2) = \epsilon$.

□

The following lemma shows that if three FIFO queues are related by $\approx_n$, applying *RemoveMark* keeps the relation intact.

Lemma 6 *For any three* FIFO *channels* $\phi, \phi_1, \phi_2 \in \Sigma^*_\phi$*, we have that if* $\phi \approx_n (\phi_1, \phi_2)$*, then* $RemoveMark(\phi) \approx_n (RemoveMark(\phi_1), RemoveMark(\phi_2))$*.* ■

Proof. Proof by induction on the number of markers $\diamond_i$ in ϕ. If there are no markers in ϕ, $\approx_n$ forbids the presence of markers in ϕ_1 and ϕ_2, hence *RemoveMark* has no effect and the property holds.

Suppose that the property holds for n markers. If a marker is added in ϕ, $\approx_n$ only requires the insertion of one or two markers in ϕ_1 and ϕ_2. Since *RemoveMark* removes that markers, the same queues are obtained as with n markers, and the property holds. □

Lemma 1 (One-split simulation). Given a well-formed $_{\mathrm{d}}\mathrm{SL}_\diamond$ program P and two distributions $D = (V_1, ..., V_n)$, $D' = (V'_1, ..., V'_n, V'_{n+1})$ of P such that $\forall i \in [1..n-1] : V_i = V'_i$ and $V_n = V'_n \cup V'_{n+1}$. We have that $[\![P]\!]_{D'}$ simulates $[\![P]\!]_D$:

$$[\![P]\!]_D \lesssim [\![P]\!]_{D'}$$

We define a relation $\mathcal{R} \subseteq \mathcal{G}^P_D \times \mathcal{G}^P_{D'}$ such that
if $G = ((\omega^G_1, \nu^G_1, \phi^G_1), (\omega^G_2, \nu^G_2, \phi^G_2), ..., (\omega^G_n, \nu^G_n, \phi^G_n), \sigma^G_1, \sigma^G_2, ..., \sigma^G_\ell, \mu^G, \xi^G)$

and
$G' = ((\omega_1^{G'}, \nu_1^{G'}, \phi_1^{G'}), (\omega_2^{G'}, \nu_2^{G'}, \phi_2^{G'}), ..., (\omega_n^{G'}, \nu_n^{G'}, \phi_n^{G'}), (\omega_{n+1}^{G'}, \nu_{n+1}^{G'}, \phi_{n+1}^{G'}),$
$\sigma_1^{G'}, \sigma_2^{G'}, ..., \sigma_\ell^{G'}, \mu^{G'}, \xi^{G'})$, $(G, G') \in \mathcal{R}$ if and only if:

1. $(\omega_i^G, \nu_i^G, \phi_i^G) = (\omega_i^{G'}, \nu_i^{G'}, \phi_i^{G'})$, $\forall i \in [1..n-1]$
2. $(\phi_n^{G'}, \phi_{n+1}^{G'}) \approx_n \phi_n^G$
3. $\forall x \in (V'_n \cup \widetilde{Var(P)} \cup OldCond(W'_n)),\ \nu_n^G(x) = \nu_n^{G'}(x)$
4. $\forall x \in (V'_{n+1} \cup \widetilde{Var(P)} \cup OldCond(W'_{n+1})),\ \nu_n^G(x) = \nu_{n+1}^{G'}(x)$
5. $(\omega_n^{G'}, \omega_{n+1}^{G'}) \succ \omega_n^G$
6. $\forall i \in [1..\ell] : \sigma_i^G = \sigma_i^{G'}$
7. $\mu^G = \mu^{G'}$
8. $\forall i \in [1..\ell] : \begin{cases} \xi^G(i) < n & \rightarrow \xi^G(i) = \xi^{G'}(i) \\ \xi^G(i) = n & \rightarrow \xi^{G'}(i) \in \{n, n+1\} \end{cases}$
   In the rest of this proof, this last property will be noted $\xi^G \equiv \xi^{G'}$

We prove that $\mathcal{R}$ is a simulation relation for $[\![P]\!]_D$ and $[\![P]\!]_{D'}$. More precisely, we prove that if $(G, G') \in \mathcal{R}$, for all $a \in (Var(P) \times \{!, ?\} \times \{\top, \bot\} \cup \{\tau\})$:

$$\forall H\ ,\ \left(G \xrightarrow{a} H\right) \implies \left(\exists H'\ ,\ G' \xrightarrow{\tau^* \cdot a} H' \wedge ((H, H') \in \mathcal{R})\right)$$

In the rest of the proof, we will use the following notation:

- $H = ((\omega_1^H, \nu_1^H, \phi_1^H), (\omega_2^H, \nu_2^H, \phi_2^H), ..., (\omega_n^H, \nu_n^H, \phi_n^H),$
  $\sigma_1^H, \sigma_2^H, ..., \sigma_\ell^H, \mu^H, \xi^H)$
- $H' = ((\omega_1^{H'}, \nu_1^{H'}, \phi_1^{H'}), ..., (\omega_n^{H'}, \nu_n^{H'}, \phi_n^{H'}), (\omega_{n+1}^{H'}, \nu_{n+1}^{H'}, \phi_{n+1}^{H'}),$
  $\sigma_1^{H'}, \sigma_2^{H'}, ..., \sigma_\ell^{H'}, \mu^{H'}, \xi^{H'})$.

Moreover, given a global state G, we note $(G)_i$ the i^{th} component of G.

The transition $G \xrightarrow{a} H$ can be derived from the following global semantic rules : Interleaving, Broadcast, Sequence Activate, Sequence Activate$'$, Sequence Assign, Sequence `IF` or Sequence `WHILE`.

[Broadcast] According to the broadcast semantic rule, if a global transition is fired, it means that one of the local processes has a `BCAST(x)` at the beginning of its workload. Let i denote that local process. We have two possibilities:

1. $i \neq n$: since $(G, G') \in \mathcal{R}$, we have that $(G)_i = (G')_i$. Since the broadcast can be fired by $(G)_i$ in $[\![P]\!]_D$, it can also be fired by $(G')_i$ in $[\![P]\!]_{D'}$. It follows directly that:

$$G' \xrightarrow{\tau} H'$$

The message is added to all FIFO channels. Therefore, FIFO channels in H and H' respect the relation $\approx_n$. None of the σ_i, μ or ξ change. It is then easy to see that $(H, H') \in \mathcal{R}$.

2. $i = n$: we have that $\omega_n^G = \texttt{BCAST(x)};\omega_n^H$. Moreover, since $(G, G') \in \mathcal{R}$, we have that $(\omega_n^{G'}, \omega_{n+1}^{G'}) \succ \omega_n^G$. Thus, by definition of $\succ$, there are two possibilities:

   (a) $\omega_n^{G'} = \texttt{BCAST(x)};\omega_n^{H'}$ and $\omega_{n+1}^{G'} = \omega_{n+1}^{H'}$, then $G' \xrightarrow{\tau} H'$ can be deduced from the broadcast semantic rule. The abstract instruction BCAST(x) is removed from $\omega_n^{G'}$, and by definition of $\succ$, we have $\omega_n^H \succ (\omega_n^{H'}, \omega_{n+1}^{H'})$. The message is added to every FIFO channel and the FIFO channels in H and H' respect the relation $\approx_n$. It is then easy to see that $(H, H') \in \mathcal{R}$ (note that all of the σ_i, as well as ξ and μ remain unchanged).

   (b) $\omega_n^{G'} = \omega_n^{H'}$ and $\omega_{n+1}^{G'} = \texttt{BCAST(x)};\omega_{n+1}^{H'}$, the case is symmetrical.

[Interleaving] According to the interleaving semantic rule, if a global transition is fired from G, it means that one of the local processes can fire a local transition. Let i denote that local process. We have two possibilities:

1. $i \neq n$: since $(G, G') \in \mathcal{R}$, we have that $(G')_i = (G)_i$. Since the local transition can be fired by $(G)_i$ in $[\![P]\!]_D$, it can also be fired by $(G')_i$ in $[\![P]\!]_{D'}$. It follows that:

$$G' \xrightarrow{a} H'$$

Only the local process i changes, the other local processes remain unchanged. Therefore, since $(H)_i = (H')_i$, we have that $(H, H') \in \mathcal{R}$.

2. $i = n$: this case must be considered more carefully. As $(G, G') \in \mathcal{R}$, we have that $(\omega_n^{G'}, \omega_{n+1}^{G'}) \succ \omega_n^G$. Let's consider each local rule separately :

   [Cycle Start] In this case, we have $\omega_n^G = \epsilon$ and $a = \tau$. Moreover, by definition of $\succ$, we have that $\omega_n^{G'} = \omega_{n+1}^{G'} = \epsilon$, thus, we can apply the cycle start rule in both $(G')_n$ and $(G')_{n+1}$. Let's

construct a transition sequence performing both start cycles from G' :

$G' \xrightarrow{\tau} G'' \xrightarrow{\tau} H'$
Where
Cycle start in local process n :
$(G')_n \xrightarrow{\tau} (G'')_n \wedge \forall i \neq n \ : \ (G')_i = (G'')_i$
Cycle start in local process n+1 :
$(G'')_{n+1} \xrightarrow{\tau} (H')_{n+1} \wedge \forall i \neq n+1 \ : \ (G'')_i = (H')_i$

Considering G' and H', we have the following equalities :

$$\nu_n^G = \nu_n^H$$
$$\nu_n^{G'} = \nu_n^{H'}$$
$$\nu_{n+1}^{G'} = \nu_{n+1}^{H'}$$

Indeed, the valuations on the variables remain the same when the cycle start rule is applied. For the FIFO channels, we have the following equalities :

$$\phi_n^H \in \mathit{Shuffle}(\phi_n^G)_\pi$$
$$\phi_n^{G''} \in \mathit{Shuffle}(\phi_n^{G'})_{\pi'}$$
$$\phi_{n+1}^{H'} \in \mathit{Shuffle}(\phi_{n+1}^{G''})_{\pi''}$$
$$\phi_n^{H'} = \phi_n^{G''} \wedge \phi_{n+1}^{G''} = \phi_{n+1}^{G'}$$

To obtain $\phi_n^H \approx_n (\phi_n^{H'}, \phi_{n+1}^{H'})$, we must choose π' and π'' as follows : let k be such that $\pi(k) = n$, and $\forall i \in [1..k] : \pi'(i) = \pi''(i) = \pi(i) \wedge \pi'(k+1) = \pi''(k+1) = n+1 \wedge \forall i \in]k+1..n+1] : \pi'(i) = \pi''(i) = \pi(i-1)$. Once the markers are in the right order, we can insert them in the right place (by selecting the right element in the *Shuffle*() sets) to obtain $\phi_n^H \approx_n (\phi_n^{H'}, \phi_{n+1}^{H'})$.
Let's look at ω_n^H : by lemma 3, we know that every input, output, and when treatment instruction inserted in ω_n^H will be inserted either in $\omega_n^{H'}$ or $\omega_{n+1}^{H'}$. More formally, this gives :

$$(Sample(Var^{in}(P) \cap V'_n, <_V), Sample(Var^{in}(P) \cap V'_{n+1}, <_V)) \succ Sample(Var^{in}(P) \cap V_n, <_V)$$

$$(Write(Var^{out}(P) \cap V'_n, <_V), Write(Var^{out}(P) \cap V'_{n+1}, <_V)) \succ Write(Var^{out}(P) \cap V_n, <_V)$$

$$(Treat(W'_n, <_W), Treat(W'_{n+1}, <_W)) \succ Treat(W_n, <_W)$$

Thus, we have $(\omega_n^{H'}, \omega_{n+1}^{H'}) \succ \omega_n^H$ and $G' \xrightarrow{\tau \cdot \tau} H'$ and $(H, H') \in \mathcal{R}$ (not that none of the σ_i changes, as is the case for μ and ξ).

[Input] In this case, we have $\omega_n^G = \texttt{INPUT(x)}; \eta_n^H$ and $a = x?v$. Then, as $(\omega_n^{G'}, \omega_{n+1}^{G'}) \succ \omega_n^G$, there are two possibilities:

(a) if $\omega_n^{G'} = \texttt{INPUT}(x); \eta_n^{H'}$ and $\omega_{n+1}^{H'} = \omega_{n+1}^{G'}$, we can apply the input semantic rule to $(G')_n$, thus we have $(G')_n \xrightarrow{a} (H')_n$. The valuation for x is modified accordingly. Thus we have $\nu_n^H = \nu_n^H[x \mapsto v]$ and $\nu_n^{H'} = \nu_n^{H'}[x \mapsto v]$. Furthermore, by definition of $\succ$, we have $(\eta_n^{H'}, \omega_{n+1}^{H'}) \succ \omega_n^H$ and by interleaving, we can perform $G' \xrightarrow{a} H'$. In conclusion, we have :

$$\begin{array}{rcl} \omega_n^H & = & \texttt{BCAST(x)}; \eta_n^H \\ \omega_n^{H'} & = & \texttt{BCAST(x)}; \eta_n^{H'} \end{array}$$

We can conclude that $(H, H') \in \mathcal{R}$ (none of the σ_i changes, which is also the case for μ and ξ).

(b) if $\omega_{n+1}^{G'} = \texttt{INPUT}(x); \omega_{n+1}^{H'}$ and $\omega_n^{H'} = \omega_n^{G'}$, the case is symmetrical.

[Message treatment] In this case, we have $\omega_n^G = MSG; \eta_n^H$, and $a = \tau$. Then, as $(\omega_n^{G'}, \omega_{n+1}^{G'}) \succ \omega_n^G$, we have:

$$\begin{array}{rcl} \omega_n^{G'} & = & MSG; \eta_n^{H'} \\ \omega_{n+1}^{G'} & = & MSG; \eta_{n+1}^{H'} \end{array}$$

We can then apply the message treatment rule or the end of message treatment rule to $(G')_n$ and $(G')_{n+1}$, we then have a sequence of transitions :

$$G' \xrightarrow{\tau} G'' \xrightarrow{\tau} H'$$

Where

$(G')_n \xrightarrow{\tau} (G'')_n \wedge \forall i \neq n : (G')_i = (G'')_i$

$(G'')_{n+1} \xrightarrow{\tau} (H')_{n+1} \wedge \forall i \neq n+1 : (G'')_i = (H')_i$

If $G \xrightarrow{\tau} H$ results from the application of the end of message treatment rule, then we may simply apply this rule to $(G')_n$ and $(G')_{n+1}$.

If the message treatment rule is applied, we have $SubCh_s(\phi_n^G) = (x, v, s) \cdot \phi''$, for some s. Since $(\phi_n^{G'}, \phi_{n+1}^{G'}) \approx_n \phi_n^G$, by lemma 5, we know that the same message can be read by the three processes,

the valuation will be modified accordingly and the list of whens will be inserted in the workload. We have :

$$\begin{array}{rcl} \omega_n^H & = & Treat(W_{n/\tilde{x}}, <_W)\eta_n^H \\ \omega_n^{H'} & = & Treat(W'_{n/\tilde{x}}, <_W)\eta_n^{H'} \\ \omega_{n+1}^{H'} & = & Treat(W'_{n+1/\tilde{x}}, <_W)\eta_{n+1}^{H'} \end{array}$$

Then, by lemma 3, we have $W_{n/\tilde{x}} = W'_{n/\tilde{x}} \cup W'_{n+1/\tilde{x}}$. It follows that

$$(Treat(W'_{n/\tilde{x}}, <_W), Treat(W'_{n+1/\tilde{x}}, <_W)) \succ Treat(W_{n/\tilde{x}}, <_W)$$

Moreover $(\eta_n^{H'}, \eta_{n+1}^{H'}) \succ \eta_n^H$, and, thus, $(\omega_n^{H'}, \omega_{n+1}^{H'}) \succ \omega_n^H$. As the modifications brought to the FIFO channels and the valuations are the same, we can conclude that $(H, H') \in \mathcal{R}$.

If the end of message treatment rule is applied, only the FIFO queues change, and `MSG` is removed from both workloads. Lemma 6 shows that $\phi_n^H \approx_n (\phi_n^{H'}, \phi_{n+1}^{H'})$.

In all the cases, we have $G' \xrightarrow{\tau \cdot \tau} H'$ and $(H, H') \in \mathcal{R}$ (Again, all of the σ_i as well as μ and ξ remain the same).

[Assignment] In this case, we have $\omega_n^G = \mathtt{x} := \mathtt{e}; \eta_n^H$ and $a = \tau$. Then, as $(\omega_n^{G'}, \omega_{n+1}^{G'}) \succ \omega_n^G$, there are two possibilities:

(a) if $\omega_n^{G'} = \mathtt{x} := \mathtt{e}; \eta_n^{H'}$ and $\omega_{n+1}^{G'} = \omega_{n+1}^{H'}$, we can apply the assignment rule to $(G')_n$, thus $(G')_n \xrightarrow{\tau} (H')_n$, and, by interleaving, we have $G' \xrightarrow{\tau} H'$. We have $\nu_n^H = \nu_n^G[x \mapsto \nu_n^G(e)]$ and $\nu_n^{H'} = \nu_n^{G'}[x \mapsto \nu_n^{G'}(e)]$. Thus, the valuation ν_n^H and $\nu_n^{H'}$ remain coherent. As x is not in the domain of $\nu_{n+1}^{H'}$, $\nu_{n+1}^{H'}$ (since the distribution is based on a partition of the variables, and each local valuation holds the values of all variables from one partition) and ν_n^H remain also coherent.

If $x \in V(P)$, we need to show that $W_{n/x} = W'_{n/x}$, that is, they share the same set of `WHEN`s (partially) conditioned on this variable. Note that, by construction of W'_n, we already have that $W'_n \subseteq W_n$ and, thus, $W'_{n/x} \subseteq W_{n/x}$. Let's suppose that $\exists w \in W_n \ : \ w \notin W'_n$. Thus, $w \in W'_{n+1}$, but w needs to access x, and the atomicity constraint on the `WHEN`'s is violated. Thus, D' would not be a distribution and we must have $W'_{n/x} = W_{n/x}$. In conclusion, we have :

$$\begin{array}{rcl} \omega_n^H & = & \mathtt{BCAST}(x); Treat(W_{n/x}, <_W); \eta_n^H \\ \omega_n^{H'} & = & \mathtt{BCAST}(x); Treat(W'_{n/x}, <_W); \eta_n^{H'} \end{array}$$

As $(\eta_n^{H'}, \omega_{n+1}^{G'}) \succ \omega_n^{H'}$, we have that

$$(\omega_n^{H'}, \omega_{n+1}^{H'}) \succ \omega_n^H$$

The case where $x \in OldCond(P)$ is trivial, as nothing is inserted in the workload : we directly have $\eta_n^H = \omega_n^H$ and $\eta_n^{H'} = \omega_n^{H'}$.
In the remaining cases ($x \in V(P)$ or $x \in OldCond(P)$), we have $G' \xrightarrow{\tau} H'$ and $(H, H') \in \mathcal{R}$ (all of the σ_i remain unchanged, as well as ξ and μ).

(b) If $\omega_{n+1}^{G'} = \texttt{x} := \texttt{e}; \eta_{n+1}^{H'}$ and $\omega_n^{G'} = \omega_n^{H'}$, the case is symmetrical.

[If] In this case, we have $\omega_n^G = \texttt{IF}\ e\ \texttt{THEN}\ \omega^\top\ \texttt{ELSE}\ \omega^\bot\ \texttt{ENDIF}; \eta_n^H$, and $a = \tau$. Then, again, as $(\omega_n^{G'}, \omega_{n+1}^{G'}) \succ \omega_n^G$, there are two possibilities:

(a) if $\omega_n^{G'} = \texttt{IF}\ e\ \texttt{THEN}\ \omega^\top\ \texttt{ELSE}\ \omega^\bot\ \texttt{ENDIF}; \eta_n^{H'}$ and $\omega_{n+1}^{G'} = \omega_{n+1}^{H'}$, we can apply the if rule to $(G')_n$, thus $(G')_n \xrightarrow{\tau} (H')_n$ and, by interleaving, we have $G' \xrightarrow{\tau} H'$. As $(G, G') \in \mathcal{R}$, we have $\nu_n^G(e') = \nu_n^G(e)$ and the same branch of the IF statement will be executed afterward. We then have :

$$\begin{array}{llll} \omega_n^H = \omega^\top; \eta_n^H & \omega_n^{H'} = \omega^\top; \eta_n^{H'} & \text{if } \nu_n^G(e) = \top \\ \omega_n^H = \omega^\bot; \eta_n^H & \omega_n^{H'} = \omega^\bot; \eta_n^{H'} & \text{if } \nu_n^G(e) = \bot \\ \omega_n^H = \eta_n^H & \omega_n^{H'} = \eta_n^{H'} & \text{if } \nu_n^G(e) = \sharp \end{array}$$

As $(\eta_n^{H'}, \omega_{n+1}^H) \succ \eta_n^H$, by definition of $\succ$, we have $(\omega_n^{H'}, \omega_{n+1}^{H'}) \succ \omega_n^H$. In conclusion, we have $G' \xrightarrow{\tau} H'$ and $(H, H') \in \mathcal{R}$, since none of the σ_i nor μ and ξ changed.

(b) if $\omega_{n+1}^{G'} = \texttt{IF}\ e\ \texttt{THEN}\ \omega^\top\ \texttt{ELSE}\ \omega^\bot\ \texttt{ENDIF}; \eta_{n+1}^{H'}$ and $\omega_n^{G'} = \omega_n^{H'}$, the case is symmetrical.

[Output] In this case, we have $\omega_n^G = \texttt{OUTPUT}(x); \omega_n^H, (a = x!v)$. Then, as usual, as $(\omega_n^{G'}, \omega_{n+1}^{G'}) \succ \omega_n^G$, there are two possibilities:

(a) if $\omega_n^{G'} = \texttt{OUTPUT}(x); \omega_n^{H'}$ and $\omega_{n+1}^{G'} = \omega_{n+1}^{H'}$, we can apply the output rule to $(G')_n$. Therefore, we have $(G')_n \xrightarrow{a'} (H')_n$ and, by interleaving, $G' \xrightarrow{a'} H'$. As $(G, G') \in \mathcal{R}$, we have $\nu_n^{G'}(x) = \nu(x)_n^G$. Thus, $a = a'$, as the same value will be given in output. As (by definition of $\succ$) $(\omega_n^{H'}, \omega_{n+1}^{H'}) \succ \omega_n^H$, we have $(H, H') \in \mathcal{R}$ (all of the σ_i, μ and ξ remain unchanged).

(b) if $\omega_{n+1}^{G'} = \texttt{OUTPUT}(x); \omega_{n+1}^{H'}$ and $\omega_n^{G'} = \omega_n^{H'}$, the case is symmetrical.

[Sequence Activate] As $(G, G') \in \mathcal{R}$, $\sigma_s^G = \sigma_s^{G'} = \epsilon$. Let i be such that $(G)_i$ fired that rule. We have two possibilities :

1. $i < n$ By definition of $\mathcal{R}$, we must have $\omega_i^G = \texttt{LAUNCH}\ s; w_i^H$. This rule can be fired from G' to obtain H'.
2. $i = n$ By definition of $\mathcal{R}$, we have two possibilities :
   - $\omega_n^{G'} = \texttt{LAUNCH}\ s; \omega_n^{H'}$. This rule can be fired from G'.
   - $\omega_{n+1}^{G'} = \texttt{LAUNCH}\ s; \omega_{n+1}^{H'}$. This case is symmetrical.

Let's consider $Exec_D(\mathcal{S}(s)))$, if $Exec_D(\mathcal{S}(s))) = \{n\}$, then by lemma 4, $Exec_{D'}(\mathcal{S}(s))) \in \{n, n+1\}$. Thus, $\xi^{H'} \equiv \xi^H$. Finally, we have that $\sigma_s^{H'} = \sigma_s^H$. In conclusion, we have that $(H, H') \in \mathcal{R}$.

[Sequence Activate'] As $(G, G') \in \mathcal{R}$, $\sigma_s^{G'} \neq \epsilon$. Let i be the process that fired this rule.

1. If $i < n$, then this rule can be fired from G' and we have that $\omega_i^H = \omega_i^{H'}$.
2. If $i = n$, then by definition of $\mathcal{R}$, we have two possibilities
   - $\omega_n^{G'} = \texttt{LAUNCH}s; \omega_n^{H'}$. This rule can also be fired from G' and $(H, H') \in \mathcal{R}$
   - $\omega_{n+1}^{G'} = \texttt{LAUNCH}s; \omega_{n+1}^{H'}$. This case is symmetrical.

[Sequence Assign] Let s be the sequence that contains the assignment, and let i be such that $\xi^G(s) = i$. First, as $(G, G') \in \mathcal{R}$, then it is easy to see that $\Upsilon_i^G(e) = \Upsilon_i^{G'}(e)$. Furthermore, we have that $\sigma_s^{G'} = \texttt{x := e};\sigma_s^{H'}$.
We have two possibilities for i

1. $i < n$ this rule can be fired from G'. As $\Upsilon_i^G(e) = \Upsilon_i^{G'}(e)$, we have that $\nu_i^H = \nu_i^{H'}$ and $\mu^H = \mu^{H'}$ Moreover, $\omega_i^{H'} = \omega_i^H$ and $\sigma_s^{H'} = \sigma_s^H$. Finally, we can easily prove that $\xi^{H'} \equiv \xi^H$
2. $i = n$, thus, $\omega_n^G = MSG; \eta_n$. By lemma 4, we have two possibilities :
   - $\xi^{G'}(s) = n$. By definition of $\mathcal{R}$, we have that $\omega^{G'} = MSG; \eta'_n$. Hence, we can fire this rule from G'. We have that $\sigma_s^H = \sigma_s^{H'}$. and, using lemma 4 we can easily prove that $\xi^H \equiv \xi^{H'}$. Remark that since the valuations (μ and ν_n are coherent between G and G', they remain so in H and H'.
   - $\xi^{G'}(s) = n + 1$. This case is symmetrical.

In conclusion $(H, H') \in \mathcal{R}$.

[Sequence IF] Let i be such that $\xi^G(s) = i$. First, as $(G, G') \in \mathcal{R}$, then it is easy to see that $\Upsilon_i^G(e) = \Upsilon_i^{G'}(e)$. Thus, in all three cases ($\top, \bot$ or $\sharp$), $\sigma^{H'} = \sigma^H$. And, using lemma 4, it is easy to see that $\xi^{H'} \equiv \xi^H$. In conclusion $(H, H') \in \mathcal{R}$.

[Sequence WHILE] Let i be such that $\xi^G(s) = i$. Since $(G, G') \in \mathcal{R}$, and $\xi^G \equiv \xi^{G'}$, it is easy to see that if $i < n$, the same process can make fire the same transition in G', and $(H, H') \in \mathcal{R}$. If $i = n$ then either process n or $n + 1$ can take the transition and update σ_s accordingly. We can therefore conclude $(H, H') \in \mathcal{R}$.

Finally, it is easy to prove that $(G_D^0, G_{D'}^0) \in \mathcal{R}$. Indeed, according to definition 14, both in G_D^0 and $G_{D'}^0$, the workloads are empty, the valuations are assigning all variables to $\sharp$ and the FIFO channel are empty. Therefore, by definition of $\mathcal{R}$ and by definition 16, we have $(G_D^0, G_{D'}^0) \in \mathcal{R}$ and :

$$[\![P]\!]_D \lesssim [\![P]\!]_{D'}$$

■

3.6 From $_\mathrm{d}$SL to $_\mathrm{d}$SL$_\diamond$

In this section, we intuitively describe the syntactical transformation from a full featured $_\mathrm{d}$SL program to a simplified $_\mathrm{d}$SL$_\diamond$ program. As said earlier, $_\mathrm{d}$SL$_\diamond$ is a subset of the $_\mathrm{d}$SL language. In this subset, we make the following restrictions: (1) `METHOD`s are supposed to be inlined, which implies that recursive calls are forbidden; (2) since no recursion is allowed, all variables outside `SEQUENCE`s can be considered to be declared globally; (3) only boolean variables are considered; (4) as we will see in this section, `METHOD LAUNCH`es and the `WAIT` instruction inside `SEQUENCE`s can equivalently be translated into code using only `WHEN`s and $\sim$. Therefore, their semantics are defined through this translation and no other semantics are given.

Removing `METHOD LAUNCH` We use the duality between the asynchronous execution of `METHOD`s using `LAUNCH` and the $\sim$ operator, in order to transform the former into code that only uses the latter.

For any `LAUNCH` instruction i starting a `METHOD M`, we introduce a new global variable $G_{i,m}$, initially set to `FALSE`. The instruction

```
LAUNCH M();
```

is then replaced by

$$G_{i,m} \texttt{ := TRUE; } G_{i,m} \texttt{ := FALSE;}$$

and a `WHEN` is used to asynchronously trigger the code of `M` as follows :

$$\texttt{WHEN } \sim G_{i,m} \texttt{ THEN M(); END\_WHEN}$$

Removing `METHOD` calls Method calls in the program text are removed by means of inlining. Note again that we can not model recursion in $_d$SL's semantics.

Removing `WAIT` instructions A `wait` instruction can only occur in a `SEQUENCE`. We do not give a formal translation of how to remove `WAIT` instructions, but give two examples which should convince that they can be translated using the concepts available in $_d$SL$_\Diamond$. The basic transformation goes as follows : (1) Cut the `SEQUENCE` into two parts, where the first part contains the `WAIT` instruction as last instruction. (2) Replace the `WAIT` instruction by a an assignment causing a raising edge on a fresh global variable (3) insert a `WHEN` in the program, which combines the condition of the `WAIT` and the newly introduced variable (4) in the body of that `WHEN`, reset the value of the variable, and start the second part of the `SEQUENCE`. This is illustrated in figure 3.7. Notice the introduction of a fresh global variable `S_part_1_WAITING`, which indicates whether or not `S` reached the `WAIT` instruction.

A more complex situation to translate happens when the `WAIT` instruction is inside some compound `IF` or `WHILE` instruction. An example for the translation in that case is given in figure 3.8, but the basic idea remains the same.

3.7 Examples

We now present three complete examples of $_d$SL programs, that have been distributed, compiled and tested with the prototype $_d$SL compiler presented in chapter 5.

3.7.1 A canal lock controller

The problem

In this example, we study the design of a controller for a canal system composed of two consecutive locks. As presented in figure 3.9, each lock is

```
SEQUENCE S ()
    // ... part1 ...
    WAIT condition;
    // ... part2 ...
END_SEQUENCE
```

```
SEQUENCE S_part1 ()
    // ... part1 ...
    S_part1_WAITING := TRUE;
END_SEQUENCE

WHEN S_part1_WAITING
     AND condition
THEN
    S_part1_WAITING := FALSE;
    LAUNCH S_part2();
END_WHEN

SEQUENCE S_part2 ()
    // ... part2 ...
END_SEQUENCE
```

Figure 3.7: Removing `WAIT` instructions from `SEQUENCE`s

```
SEQUENCE S ()
    // ... part1 ...
    WHILE cond DO
       // ... part 2 ...
       WAIT condition;
       // ... part 3 ...
    END_WHILE
    // ... part4 ...
END_SEQUENCE
```

```
SEQUENCE S_part1 ()
    // ... part1 ...
    LAUNCH S_part_2();
END_SEQUENCE

SEQUENCE S_part2 ()
    IF cond THEN
      // ... part2 ...
      S_part2_WAITING := TRUE;
    ELSE
      LAUNCH S_part4();
    END_IF
END_SEQUENCE

WHEN S_part2_WAITING
     AND condition THEN
     S_part2_WAITING := FALSE;
     LAUNCH_S_part3();
END_WHEN
SEQUENCE S_part3()
    // ... part 3 ...
    LAUNCH S_part2 ();
END_SEQUENCE

SEQUENCE_S_part4()
    // ... part 4 ...
END_SEQUENCE
```

Figure 3.8: Removing `WAIT` instructions from within `WHILE` inside `SEQUENCE`s

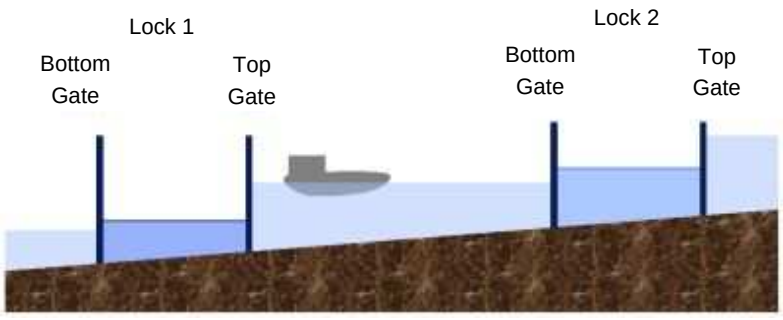

Figure 3.9: Canal locks

composed of two gates, a top and a bottom one. In between the top and the bottom gates of each lock, the water level can be controlled (i.e. the inside of a lock can be filled or emptied). The different commands of this system (opening/closing a gate, emptying/filling a lock) can be accessed via push buttons on a control panel. For this system to function properly, several constraints must be satisfied:

1. two consecutive gates cannot be opened at the same time
2. a gate can only be opened if the water level on each side is the same
3. the water level inside a lock can only be changed if both its top and bottom gates are closed

The purpose of the controller is to ensure that the previous constraints are verified at all time. Whenever a command is introduced via the control panel, before taking the appropriate action, the controller must first check that it will not jeopardize the system, in which case, the action is not taken, and a red light on the control panel is switched on to indicate that the command is forbidden.

The solution

We use the following global variables to model the two locks :

```
CLASS GATE
    motor_command : INT;
    closed, opened, motor_direction,
    button_open, button_close,
    order_given : BOOL;
```

```
END_CLASS

CLASS LOCK
    bottom_gate, top_gate : GATE;
    water_command : INT;
    water_down, water_up,  water_direction,
    button_fill, button_empty : BOOL;
END_CLASS

GLOBAL_VAR
    lock1, lock2 : LOCK;
    not_allowed_led : BOOL;
END_VAR
```

The idea to implement the controller in dSL is the following. Whenever an order is given, a corresponding boolean variable `order_given` is set (there is an `order_given` variable for each gate and one for the water level of each lock). When receiving a command, the controller has to check that all the requirements are satisfied and, using those `order_given` variables, that no order on the checked gates and water levels are given (note that an order to close a gate can never violate a constraint). The `order_given` variables are, of course, reset when an order is completed. In this implementation, each command is monitored by a `WHEN` construct. As an example, figure 3.10 presents the `WHEN` monitoring the command "open the bottom gate of lock2" (the complete dSL source can be found in appendix C).

Note that for all the `order_given` variables, the '~' operator cannot be used. For example, in figure 3.10, if `lock2.top_gate.order_given` was tilded, when an order is given to open the bottom gate of lock2, the controller would check if `~lock2.top_gate.order_given` is false. However, in that case, because of communication delays, an order might have been given. The controller would then allow the bottom gate of lock2 to open while the top gate is ordered to open, which leads to a violation of the given constraints.

Distribution

In this example, the maximal distribution is composed of 9 sites. We graphically represent the distribution constraints induced by definition 1 by a graph where nodes are either `WHEN`s or global variables. Each time a variable appears in a `WHEN`, there is an edge between that variable and the `WHEN`. It is easy to show that variables in connected components of this graph must be in the same distribution, to get a correct distribution. Furthermore, the set

```
WHEN lock2.bottom_gate.button_open THEN
  IF (~lock2.top_gate.closed) AND (not lock2.top_gate.order_given) AND
     (~lock1.top_gate.closed) AND (not lock1.top_gate.order_given) AND
     (~lock2.water.down) AND (not lock2.water_order_given)
  THEN
    not_allowed_led := false;
    lock2.bottom_gate.order_given := true;
    LAUNCH lock2.bottom_gate<-open();
  ELSE
     not_allowed_led := true;
  END_IF;
END_WHEN
```

Figure 3.10: `WHEN` monitoring the command "open the bottom gate of lock2"

of variables in the connected components defines the maximal distribution. Figure 3.11 shows this graph for the current example, where the uppercase letters indicate the different sites. Circle nodes indicate the different `WHEN`s present in the program (which can be consulted in appendix C), while the boxes in the figure represent the variable nodes. The solid lines represent the appearance of a variable in a `WHEN`. Note that on this representation, some edges are dotted to indicate the use of tilded variables which break the atomicity constraints and thus allow a more liberal maximal distribution than the one obtained without them.

The maximal distribution is composed of :

Site	Variables	`WHEN`s
A	`lock1.top_gate.{motor_command,opened,closed}`	9

B	`lock{1,2}.order_given`
	`lock{1,2}.{top,bottom}_gate.button_open`
	`not_allowed_led`
		1,2,3,4

C	`lock1.bottom_gate.{motor_command,opened,closed}`	10
D	`lock2.bottom_gate.{motor_command,opened,closed}`	12
E	`lock2.top_gate.{motor_command,opened,closed}`	11
F	`lock1.bottom_gate.button_close`	5
G	`lock1.top_gate.button_close`	6
H	`lock2.bottom_gate.button_close`	7
I	`lock2.top_gate.button_close`	8

The actual distribution used in practice for this example contained three sites : $\{A \cup C, D \cup E, B \cup F \cup G \cup H \cup I\}$. This distribution is a natural 3-site distribution, where one controller is used for each lock, and where a third controller is connected to the control panel in order to let the operator give orders to the system.

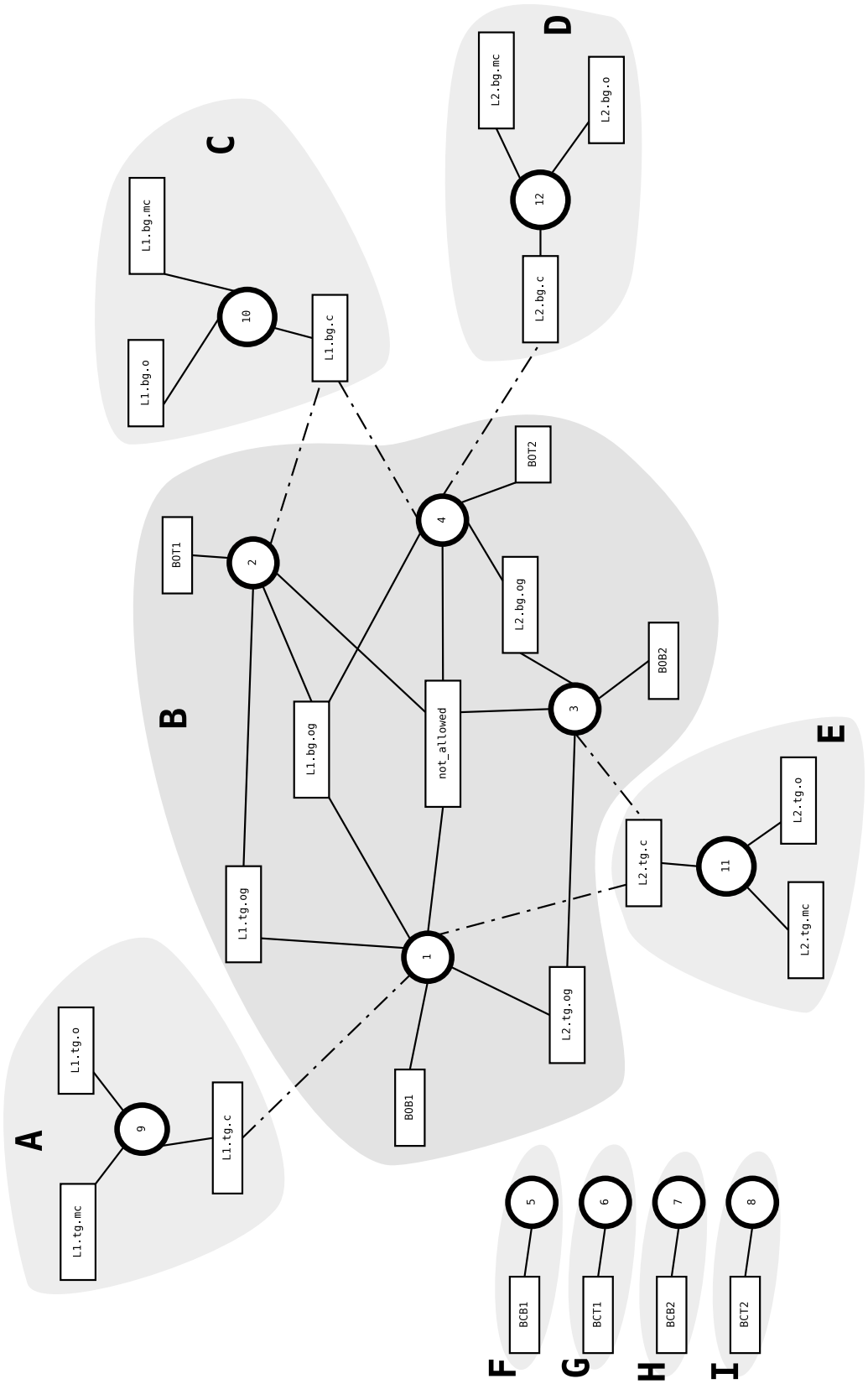

Figure 3.11: Maximal distribution of the locks controller

3.7.2 A Conveyor belt

The problem

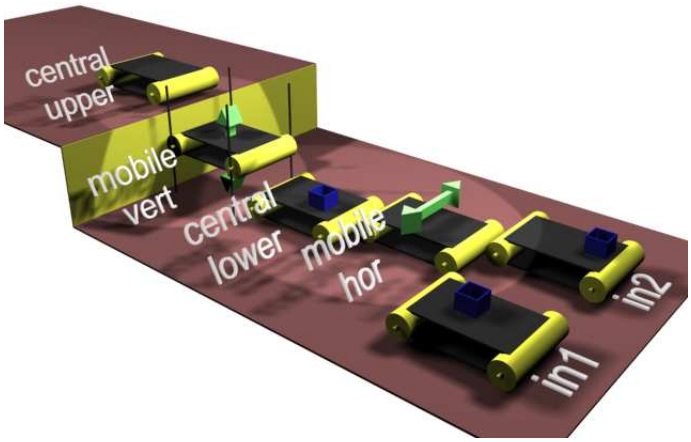

Figure 3.12: A conveyor belt

In this application, boxes have to be moved from one side of a plant to the other. The departure point of the plant is on a lower level than the arrival point. An additional problem is that boxes arrive from two different sources while there is only one transport belt that allows to lift the boxes from the lower floor to the upper floor. For this reason, two of the belts are mobile. There is one elevator belt, which can go up and down, in order to move packages from the lower floor to the upper floor. The other mobile belt can move from the left to the right and back, to allow packages to move from the different sources to the central belt. The complete setup is represented in figure 3.12.

The problem encountered here is very similar to the problem faced in the locks controller. However, the major difference here is that the system is completely standalone, in the sense that there is no operator who supervises the system and has to issue commands. We also extensively use `SEQUENCE`s in this solution, which is not the case in the previous example.

All restrictions on the possible behaviors of the belts can be summarized in one constraint : if there is a package at the end of a belt, the belt should stop unless the next belt is in front of it and is running. In the case where no next belt is present, boxes would fall on the floor. In the other case, where the next belt is stopped, either the boxes or the belts would get damaged.

The solution

To implement this example, we used the following global variables :

```
CLASS Conveyor_belt
    motor       : INT;   // Linked to the belt's engine
    box_at_end  : BOOL;  // TRUE when a box is detected
END_CLASS

CLASS Mobile_belt
    motor       : INT;   // Linked to the lateral engine
    in_pos1     : BOOL;  // TRUE when in position 1
    in_pos2     : BOOL;  // TRUE when in position 2
    belt        : Conveyor_belt;
END_CLASS

GLOBAL_VAR
    belt_in1, belt_in2,
    belt_lower_central,
    belt_upper_central        : Conveyor_belt;
    mobile_hor, mobile_vert   : Mobile_belt;

    mobile_hor_free           : BOOL; // TRUE when no box is
                                // on the mobile horizontal belt
END_VAR
```

Some simple methods involve the activation/deactivation of the belts :

```
METHOD Conveyor_belt::go()
    self.motor := 20;
END_METHOD

METHOD Conveyor_belt::stop()
    self.motor := 0;
END_METHOD

METHOD Mobile_belt::go1()
    IF (NOT self.in_pos1) THEN
       self.belt<-stop();
       self.motor  := -40;
    END_IF
END_METHOD // Analogue for go2

METHOD Mobile_belt::stop()
    self.motor := 0;
END_METHOD
```

Next, some WHENs shut down the engines of the belts when needed. Remark how convenient the WHEN IN structure is to express this global condition.

```
// The entry belt should always be running, unless a box is
// at its end
WHEN NOT belt_in1.box_at_end THEN belt_in1<-go() END_WHEN
// The same goes for belt_in2

// When a box arrives at the end of a belt, stop that belt
WHEN IN Conveyor_belt self.box_at_end THEN
   self<-stop();
END_WHEN

// The mobile belts should stop moving when they reach one of their
// ending positions
WHEN IN Mobile_belt self.in_pos1 OR self.in_pos2 THEN
   self<-stop();
END_WHEN
```

Now, we give three SEQUENCEs that are used to move the boxes from the two sources to the target. These SEQUENCEs are started using some WHENs that are given further on.

```
SEQUENCE belt_in1_to_lower_central()
    mobile_hor<-go1();

    WAIT mobile_hor.in_pos1;

    // Mobile belt is now in front of belt_in1
    // Start the two belts
    mobile_hor.belt<-go();
    belt_in1<-go();

    // Wait for the box to be on mobile_hor
    WAIT mobile_hor.belt.box_at_end;

    // Start both belts. The box will leave
    // the mobile_hor belt
    belt_lower_central<-go();
    mobile_hor.belt<-go();

    // Wait for the box to be on lower_central
    WAIT belt_lower_central.box_at_end;

END_SEQUENCE  // Analogue for belt_in2_to_lower_central
```

```
SEQUENCE lower_central_to_upper_central()
    // Wait for the elevator to be down
    WAIT mobile_vert.in_pos1;

    mobile_vert.belt<-go();
    lower_central<-go();

    WAIT mobile_vert.belt.box_at_end;

    mobile_vert<-go2();

    // Wait for the elevator to be up
    WAIT mobile_vert.in_pos2;

    belt_upper_central<-go();
    mobile_vert.belt<-go();

    // Wait for the box to leave the elevator
    WAIT belt_upper_central.box_at_end;

    mobile_vert<-go1();
END_SEQUENCE
```

Finally, here are the `WHEN`s that put everything together.

```
WHEN ~belt_in1.box_at_end
     AND mobile_hor_free
THEN
     mobile_hor_free := FALSE;
     LAUNCH belt_in1_to_lower_central();
END_WHEN // Analogue for belt_in2

WHEN ~belt_in1_to_lower_central.ENDED
     mobile_hor_free := TRUE;
END_WHEN // Analogue for belt_in2

WHEN belt_lower_central.box_at_end
     AND lower_central_to_upper_central.ENDED
THEN
     LAUNCH lower_central_to_upper_central();
END_WHEN
```

There are some interesting points to observe in this example. First of all, we make the hypothesis that boxes arrive on `belt_in1` and `belt_in2` with intervals that are larger than the time needed for a box to traverse one belt.

In that case, the variable `mobile_hor_free` lets only one box pass from the source to the central belt by means of the `SEQUENCE`s `belt_in{1,2}_to_lower_central()`. Indeed, observe also that this variable is governed by two `WHENS` that must be on the same site, which results in the desired property.

Next, observe that the auto-generated variables `.ENDED` are used to make sure that no `SEQUENCE` is `LAUNCH`ed until it ended its execution.

In the case where both sequences `belt_in{1,2}_to_lower_central`, are competing to get a box to the central lower belt, only one box will be allowed to pass. Indeed, atomic code is used to govern the variable `mobile_hor_free`, and therefore only one of both `WHEN`s on that variable will be triggered, launching the corresponding sequence.

3.7.3 A railway system

The Problem

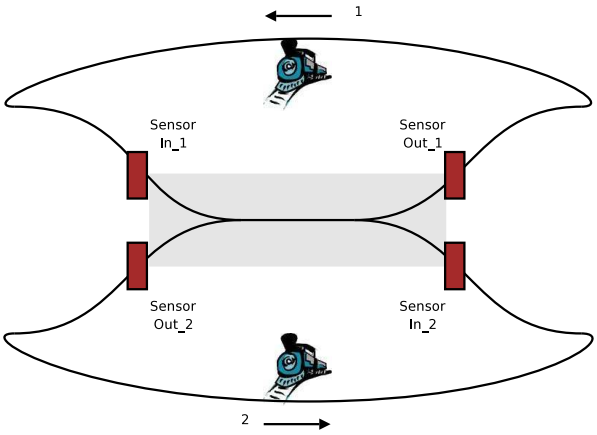

Figure 3.13: A railway system

In this example, two trains arriving from different directions travel for some distance over the same track. The example here is much shorter than the previous ones, but has the benefit of showing exactly what is meant by the atomic constraints, and how the programmer is confronted with it. The configuration for the two trains is shown in figure 3.13.

The problem is obvious : at no time, both trains should be on the central track at the same time.

The solution

The simplest way to handle this problem is to make the trains stop when they want to engage on the central track. Next, the trains ask for the permission to enter the critical section, and wait until that permission is granted. We chose this solution and implement it in dSL.

We have the following global variables :

```
CLASS Train
   motor       : BOOL; // TRUE: train running, FALSE: train stopped
   id          : INT;  // 1 for train 1, 2 for train 2
   enter, exit : BOOL; // Sensors, TRUE when entering/leaving
                       // the critical section
END_CLASS

CLASS CriticalSection
   waiting_1 : BOOL; // Train 1 is waiting
   waiting_2 : BOOL; // Train 2 is waiting
   flag      : INT;  // 0  = Central Track free
                     // 1  = Train 1 on Central Track
                     // 2  = Train 2 on Central Track
END_CLASS

GLOBAL_VAR
   train_1, train_2 : Train;

   cs : CriticalSection;
END_VAR
```

Next, we have some methods to start and stop the trains :

```
METHOD Train::stop()
    self.motor := FALSE;
END_METHOD
METHOD Train::go()
    self.motor := TRUE;
END_METHOD
```

In addition, the critical section is specified as follows :

```
METHOD CriticalSection::ask_for_1()
    self.waiting_1 := TRUE;
END_METHOD
METHOD CriticalSection::ask_for_2()
    self.waiting_2 := TRUE;
END_METHOD
```

```
METHOD CriticalSection::leave()
    self.flag := 0;
END_METHOD
WHEN cs.waiting_1 AND cs.flag == 0 THEN // WHEN 1
    cs.waiting_1 := FALSE;
    cs.flag := 1;
END_WHEN
WHEN cs.waiting_2 AND cs.flag == 0 THEN // WHEN 2
    cs.waiting_2 := FALSE;
    cs.flag := 2;
END_WHEN
```

Finally, the reaction on the events produced by the entrance and exit sensors for train 1 is specified by the following `WHEN`s (replacing 1 by 2 gives the code for `train_2`:

```
WHEN train_1<-enter THEN     // WHEN 3 (3' for in_2)
     train_1<-stop(); LAUNCH cs<-ask_for_1();
END_WHEN
WHEN IN Train self.exit THEN // WHEN 4  for train_1
     LAUNCH cs<-leave();     // WHEN 4' for train_2
END_WHEN
WHEN IN Train ~cs.flag = self.id THEN // WHEN 5  for train_1
     self<-go();                      // WHEN 5' for train_2
END_WHEN
```

The critical section is respected by this implementation, since the trains immediately stop before entering the central track (`train{1,2}<-stop();`) and wait for permission to proceed (`WHEN  cs.flag = {1,2}`). What is crucial in this example, is that the permission to enter the critical section, which is asked using `LAUNCH cs<-ask_for_{1,2}`, is handled in an atomic manner. Indeed, the different `WHEN`s on the variables of `cs` must all be located on a single site, which means that there will be no concurrency involved during these decisions.

Once access is granted, the train will be notified through the tilded copy of `cs.flag`, and when it sees the right value, it can start moving again. It is not possible that a train reenters the central track, without having asked for, and received permission. Indeed, the rising edge semantics on the `WHEN`'s condition ensures that the value of `flag` must change before the train can reenter its critical section. Once the value of `flag` changes, it will only come back to the trains' id after it asked for permission, and was granted access.

Notice that there is no fairness in the controller. The triggering conditions of the different `WHEN`s are evaluated in the order in which the `WHEN`s

are declared in the program text. If train 1 and train 2 approach the critical section at the same time, train 1 will be granted access.

Distribution constraints

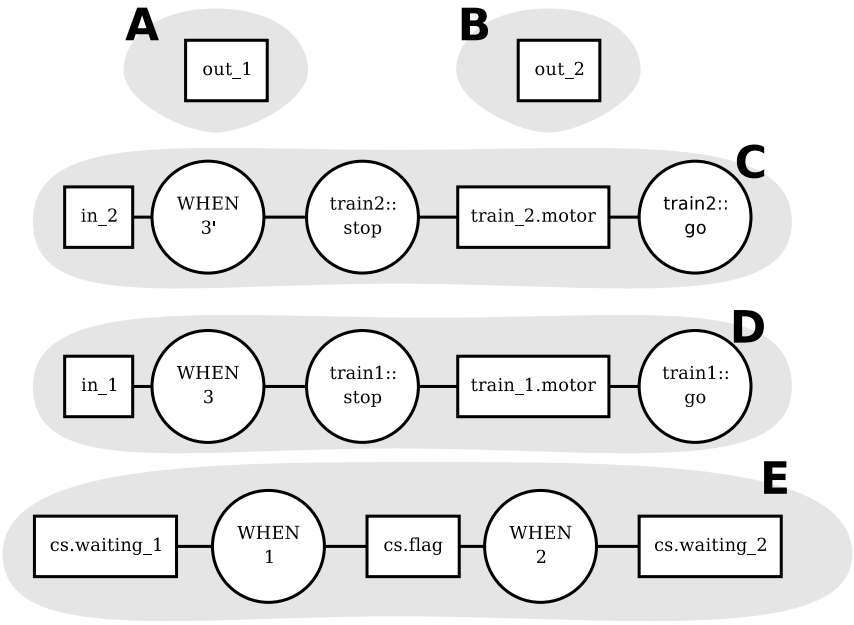

Figure 3.14: Maximal distribution for the train controller

A summary of the graphical representation for the maximal distribution can be found in figure 3.14. First of all, we can clearly see that the instructions which manage the critical section have to be on the same site, as explained before. Next, one can see that the maximal distribution consists of 5 sites, which are marked with capital letters in figure 3.14. The real interesting point however, is that the sensor which captures the train's entrance onto the central track has to be on the site of the train's engine, due to the constrains of `WHEN` 3 and $3'$, if not, the controller is not distributable.

The whole point here is that due to communication delays (or maybe worse, communication failure), the critical section cannot be guaranteed if the sensor `in_1` is not located on the site of the train. Indeed, if the site with the sensor can only observe the presence of a train, but can only make the train stop using slow or defectuous communications, the critical section cannot be guaranteed.

So, if the programmer has a distribution in mind where `in_1` is distributed on a separate site, the distributer will signal the problem by show-

ing that there is a `WHEN` (3) and a `METHOD` (`train_1<-stop()`) through which atomic control may flow from `in_1` to `train1.motor`. The programmer, then has different possibilities, which consist in using a $\sim$, a `LAUNCH` or changing the physical distribution of actuators and sensors. Here clearly, the distributer signals an important issue, saying that if the sensor and actuator are on different sites, the program cannot assure correct behavior, under the hypothesis that network delays may be unbounded, which is the basic hypothesis made in $_d$SL. Therefore, the sensor has to be on the site of the engine.

In the case where a correct distribution is chosen, and communications fail to work or are not fast enough, the train will come to a complete halt, and the trains will wait before until the communications are functioning correctly. Note that in this case, this is the best possible action to take in case of communication failure between the different entities in the system.

Generalization

It is straightforward to extend this controller in such a way that it handles n trains. One can easily add the following `WHEN`s to the program to handle the sensors of the trains (where `ID` is changed by the train's identifier) :

```
WHEN in_ID  THEN
     train_ID<-stop(); LAUNCH cs<-ask_for_ID();
END_WHEN
```

The management of the critical section can be generalized as follows :

```
METHOD CriticalSection::ask_for_ID()
    self.waiting_ID := TRUE;
END_METHOD
WHEN cs.waiting_ID AND cs.flag == 0 THEN
    cs.waiting_ID := FALSE;
    cs.flag := ID;
END_WHEN
```

Chapter 4

dSL's Distribution

In this chapter, we study the algorithmic difficulties that are encountered with the automatic code distribution of dSL. We show that the problem of checking the atomicity constraints is easy (i.e. deciding whether a given dSL program with an associated localization table is distributable or not can be done efficiently). However, the problem of generating optimal code is shown to be equivalent to the multiterminal cut problem, a well-known NP-Complete problem, for which we propose a new and efficient heuristic. The main results in this chapter were also presented in [DeWGM05].

4.1 Localizing instructions, a coloring problem

In this chapter, we abstract the problem of assigning variables and instructions to sites to a coloring problem on graphs. There is a one-to-one mapping between the colors and the sites. We do this, because the problem of producing optimal code can be shown (after simplification) to be equivalent to the multiterminal cut problem.

The assignment of instructions and variables to the set of sites is basically performed in two steps. Conceptually, the distributer checks the atomic constraints, and colors atomic code first. This results in an incomplete assignment, and therefore, a second step colors all remaining instructions and variables resulting from sequential code. In practice, the distributer combines the first two steps using the algorithm presented in section 4.3.

In section 4.2, we first present the problem of checking the atomic constraints. In section 4.3 we study the problem of localizing sequential instructions, with respect to a certain performance criterion. This performance criterion is based on the number of times each instruction is executed. Although this information cannot be computed, we assume a sufficiently ac-

curate estimation is provided. This can be done for example using profiling or monitoring tools. In section 4.5 we study how instructions might be reordered in order to increase the performance during execution.

4.2 Localizing atomic instructions

4.2.1 Informal presentation

The localization of atomic instructions can be seen as the coloring of vertices in an undirected graph. Informally, vertices in this graph are either instructions or global variables, where some of the variables have a fixed color : this is the case for all external variables present in the localization table, since each such variable will be colored with the color of the corresponding site. Edges in the graph represent the atomic constraints: each instruction has an edge to all of the global variables present in the instruction, while edges between instructions are inserted when atomic control may flow from one instruction to another. The resulting graph is then used in a straightforward manner to color all instructions and variables, or to reject the program if it is not distributable. Indeed, each connected component in the graph must be of the same color since atomic control may flow from any instruction in the component to another, and instructions must have their variables present to be executed. If two vertices in a connected component have different imposed colors, the program is not distributable.

4.2.2 Formal definitions

Definition 18 (Localization table) *A localization table T for a dSL program P is a total assignment from the set of external variables to the set of sites : $T \in Var^{in}(P) \cup Var^{out}(P) \mapsto S$, where S is the set of sites (one can think of S as $[1..k]$ where k is the number of sites).* ♦

Definition 19 (Compatible coloring) *A coloring c is a mapping from the set of instructions and variables to the set of sites. A coloring c is compatible with a localization table T if for all external variables $x \in Var^{in}(P) \cup Var^{out}(P) : c(x) = T(x)$.* ♦

The term *instructions* in the following definition denotes both simple instructions such as assignments and function calls, as well as the parts of compound instructions such as `IF` and `WHILE`. For an `IF`, there is an instruction evaluating its condition, and a list of instructions for both its branches. For a `WHILE`, there is an instruction for the evaluation of its

condition and a list of instructions for its body. To take into account the global variables which appear in the condition of a `WHEN`, an instruction here also designates the evaluation of the condition of a `WHEN`.

Definition 20 (Synchronous control flow) *An instruction i' is reachable through synchronous control flow from i, noted $i \rightsquigarrow_a i'$ if one of the following conditions holds*

- *i and i' are consecutive in the same* `WHEN` *(this includes instructions in the body of the* `WHEN` *and the evaluation of the condition of the* `WHEN`*)*
- *i' is the condition of a* `WHEN` *that might be immediately triggered by the execution of i, i.e. i affects a non tilded variable which appears, or may be referenced to, in the i' condition of a* `WHEN`
- *i' is the first instruction in a method called by i without* `LAUNCH`

♦

Definition 21 (Atomic Color Graph) *The atomic color graph G_a is a graph $G_a(V, E, T)$ associated to a dSL program P and a localization table T, where $V = \{i | i$ is an instruction in $P\} \bigcup Var(P)$ is the set of vertices, and $E \subseteq \{\{v, v'\} | v, v' \in V\}$ the set of edges which is characterized as follows :*

- $\forall x \in Var(P), \forall i \in instr(P) : x \in used(i) \Rightarrow \{x, i\} \in E$
- $\forall i, i' \in instr(P) : i \rightsquigarrow_a i' \Rightarrow \{i, i'\} \in E$

♦

We do not give a formal definition of $used(i)$, but remark that if i involves an array access, or `self`, the entire set of variables that can be referenced to by these dynamic constructs is also included in $used(i)$.

With these definitions, we can formally state the atomic coloring problem.

Definition 22 (Atomic Coloring Problem (ACP)) *The atomic coloring problem consists in, given an atomic color graph G_a, finding a mapping $c : V \mapsto \mathcal{S}$ compatible with T such that for all connected components C in G_a :*

$$n, n' \in C \Rightarrow c(n) = c(n')$$

♦

Remarks

The definition of G_a and its connected components can easily be matched with definition 1 of Distribution, given in the previous chapter. The only difference here is that we use a coloring on instructions and variables, while the formal definition of distribution involves only variables (and thus `WHEN`s). Note that, in contrast to the definition of G_a, definition 1 does not have to take into account the `LAUNCH`, or `METHODS`, since those are not defined in ${}_{\mathrm{d}}\mathrm{SL}_{\diamondsuit}$.

Note that all instructions are used in the definition of G_a, since all instructions (even instructions inside `SEQUENCE`s) must be able to execute in an atomic manner. The fact that an instruction `x := y`, where `x` and `y` are global variables, is present in a `SEQUENCE` still forces `x` and `y` to be on the same site.

Localizing the remaining atomic instructions

We already presented examples where a complete consistent coloring can not be found, but note that this coloring, if it exists, is not necessarily unique. Consider for example the following snippet of code :

```
WHEN ~x > 10 THEN
     LAUNCH M();
END_WHEN
```

Here, clearly there is no localization constraint on this atomic code, and it may therefore be located on any site.

In practice, the number of sites is fixed by the localization table. And these kind of codes must therefore be distributed amongst the sites defined in the localization table. If all such codes end up on the same site, that site might get too much load compared to the other sites. In such cases, we use a load balancing criterion to decide where to put such codes. We therefore count the number of instructions on each site, and assign the remaining chunks of atomic code in such a way that the maximum number of instructions on any site is kept minimum.

Calculating the optimal distribution, minimizing the maximum number of instructions on any site, of these remaining atomic codes is hard, since it requires to solve the NP-Complete Minimum Makespan problem [LKB77]. This problem can be formally stated as follows.

Definition 23 (Minimum Makespan) *Given a set J of n jobs, and for each job a duration $w_i \in \mathbb{Z}^+, i \in [1..n]$. The Minimum Makespan problem consists in an assignment of all jobs to m identical machines, such that*

the completion time is minimum, i.e. find a partition of J in m subsets $J_1, ..., J_m$, such that $max_i \sum_{k \in J_i} w_k$ is minimum. ♦

This is the bad news. The good news is that there exists a simple factor 2 approximation algorithm [Gra66], which consists of randomly picking one job, and scheduling it on the machine having the least weight.

Algorithm

The algorithm that constructs G_a (cfr. fig 4.1) and checks the atomic constraints can be made very efficient. If a hashtable is used to hold the vertices of G_a, the creation or retrieval of a vertex (`add_vertex`) and edge creation (`add_edge`) can be done with $O(1)$ average complexity and $O(N)$ worst case complexity.

If, together with this dynamic structure, a one-pass compiler is used to parse the input dSL program, the creation of G_a can be done with N calls to `parser_gettoken`, resulting in an average complexity in $O(N)$, where N is the number of variables and instructions.

Once G_a is constructed, checking the atomic constraints can be combined with the atomic coloring by simple traversal of the vertices in the graph. This can be done with complexity in $O(|V| + |E|)$.

Therefore, we can conclude that the atomic coloring problem can be solved with average complexity in $O(|V| + |E|)$.

Remark that the algorithm cannot handle forward calls. A simple solution to this problem consists in scanning the program text once to obtain all `METHOD`s and then running the algorithm as presented. Note that this does not alter the $O(|V| + |E|)$ complexity.

Running example

Let us take a look at our running example, introduced in figure 3.5 from section 3.2. Figure 4.2 gives a representation of G_a associated to this program, before atomic coloring. Circled vertices represent instructions (here the line number is used for conciseness), while boxes are used to represent global variables.

Here, the localization table assigns `led` and `alarm` on one site (which is the site of the control panel), `temperature` and `heater.state` are located on another site (the site of the heater).

Since there are no connected components in G_a with different imposed colors, we can conclude that the program is distributable, with the resulting

```
prev_instr     = nil;
in_atomic_code = FALSE;

while((token=parser_gettoken()) != END) {
      if (is_when(token)) {
            in_atomic_code = TRUE;
            // Creation or retrieval of a vertex for token
            prev_instr = add_vertex(token);
            for each gvar = non tilded global var in condition(token) {
                x = add_vertex(gvar);
                add_edge(prev_instr,x);
            }
      } else if (is_end_when(token) or is_end_method(token)) {
            in_atomic_code = FALSE;
            prev_instr = nil;
      } else if (is_method(token) and is_marked(atomic(token)) {
            in_atomic_code = TRUE;
      } else if (in_atomic_code && is_instruction(token)) {
            // Create a vertex for this instruction
            i = add_vertex(token);

            // Create an edge with the previous instruction
            if (prev_instr) {
                  add_edge (prev_instr, i);
            }

            // Create edges to all global variables in the instruction
            for each gvar = non tilded global var in token {
                x = add_vertex(gvar);
                add_edge(i,x);
            }

            // Create edges to all WHENs that may get triggered
            for each WHEN w such that w can be triggered by i {
                add_edge (i, condition(w));
            }

            // Handle synchronous calls
            if (token == CALL and not LAUNCH)  {
                  add_edge(token, first(called(token)))
                  mark_atomic (called(token));
            }
            prev_instr = i;
      }
}
```

Figure 4.1: Construction of G_a.

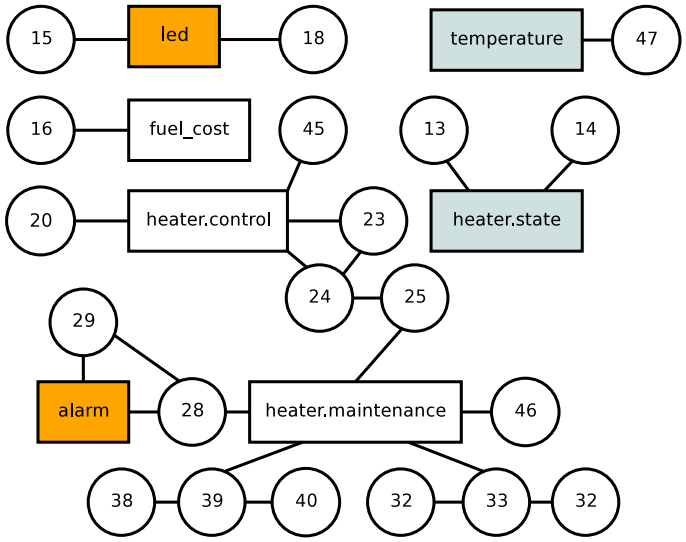

Figure 4.2: G_a, T for the heater example.

atomic coloring depicted in figure 4.3. Note that `fuel_cost` and instruction 16 are not colored after this phase. How this instruction is colored, is explained in the next section.

4.3 Localizing sequential instructions

In this section, we study how the remaining sequential instructions are colored by the dSL distributer. The main idea used to color these instructions is performance-related. Indeed, since only sequential instructions remain, they can be located on any site. However, we show that some colorings are more interesting than others, since they result in fewer messages during execution, which improves the performance of the control system.

4.3.1 Informal presentation

Consider our running example of figure 3.5, after atomic coloring, as in figure 4.3. The only uncolored instruction is defined on line 16 and involves the global variable `fuel_cost`. If we look closely at the surrounding instructions, we can see that the previous instruction `led := TRUE;` is localized on a certain site, and that the next instruction is localized on the same site. In section 3.2.1, we briefly argued that between each pair of instructions in a `SEQUENCE` that are localized on different sites, the distributer has to insert extra synchronization code which results in synchronization messages during

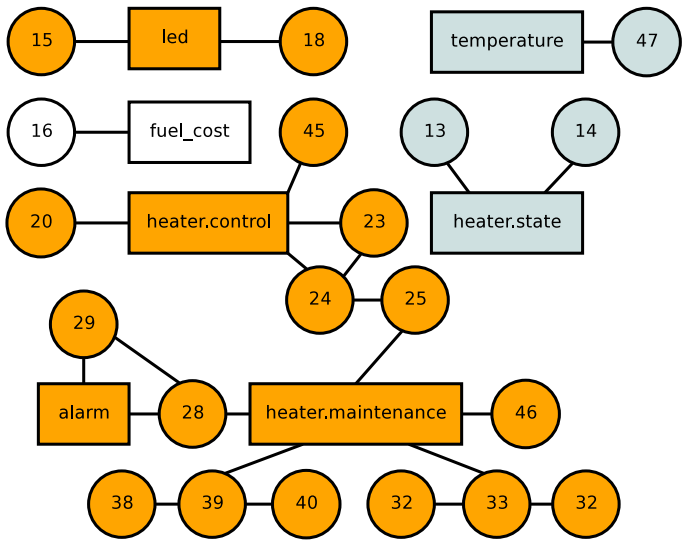

Figure 4.3: G_a for the heater example, after atomic coloring.

execution.

Clearly, if we assign `fuel_cost` and instruction 16 to the same site as its preceding and successor instruction, there will be no such additional synchronization code, resulting in no additional messages during execution. If we choose to assign the instruction and the global variable to any other site, extra code will be generated, and more messages will be needed during execution. The code clearly is more efficient when the first choice is made.

In order to evaluate the number of expected messages associated to a certain coloring, we must be able to evaluate the expected number of times control flows between the different instructions. This can be complex in cases involving `IF` and `WHILE` instructions. Once this information is known, the sequential coloring problem consists in finding a coloring such that the number of expected messages during execution is minimum. The expected number of times control flows through an instruction is defined as weight on the control flow edges between instructions, and is recursively defined using the grammar of dSL. For each instruction i, this weight is the sum of the weight inflicted by i and the weights induced by the instructions contained in i.

We first give an intuitive presentation of weighted control flow, through a graphical representation, based on the syntax of simple and compound instructions. We show how the weighted control flow is calculated in the case of a list of instructions, the `IF` case, the `WHILE` case and in the case of a simple instruction like an assignment. In the next section, we give a formal

definition in terms of an attribute grammar.

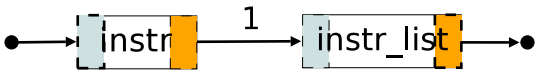

Figure 4.4: Weighted control flow for a list of instructions.

Figure 4.4 graphically represents the case of a sequence of instructions. Such a sequence takes the syntactical form *instr_list* ::= *instr* **";"** *instr_list*. Each instruction has a first part, which is marked in blue, and a last part, marked in yellow. The first and last part of a given instruction may be different in the case of compound instructions, but will be the same for simple instructions such as assignment.

The definition of the first and last part of the instruction list are surrounded in thick dotted rectangles. In the case of the instruction list, the first part is defined as the first part of the first instruction in the list. For example, if the first instruction in the list is an `IF` instruction, the first part of the instruction list will be the first part of the `IF` instruction, which is defined as the evaluation of the `IF`'s condition. Similarly, the last part of the instruction list is defined as the last part of the last instruction in the list.

The additional weighted control flow introduced by the use of an instruction list has weight 1 and is defined between the last part of the first instruction, and the first instruction of the second part of the list, since the list makes control flow between those two instructions. To take into structure of the list's sub-elements (i.e. an instruction and another list), we define the total weighted control flow induced by a list of instructions as this additional weighted control flow and the total weighted control flow induced by both parts of the list.

Figure 4.5 represents the case of an `IF` instruction, which is syntactically defined as *instr* ::= **"IF"** *expr* **"THEN"** $instr\_list_1$ **"ELSE"** $instr\_list_2$ **"END_IF"**. Again, the thick dotted lines define the first and last part of the compound `IF` instruction. In the case of an `IF`, its first part is the evaluation of its expression, the last is defined as the `END_IF` part of the instruction. The additional weighted control flow has weight $p - \frac{\psi}{2}$ between the evaluation of the condition and the first instruction from the `TRUE` branch, and from the last instruction of the `TRUE` branch to the `END_IF`. Note that ψ denotes the probability that the condition evaluates to `UNKNOWN`, hence this weight on the control flow between the evaluation of the expression and the `END_IF`. Similarly, control flow with weight $1 - p - \frac{\psi}{2}$ is defined for the `FALSE` branch

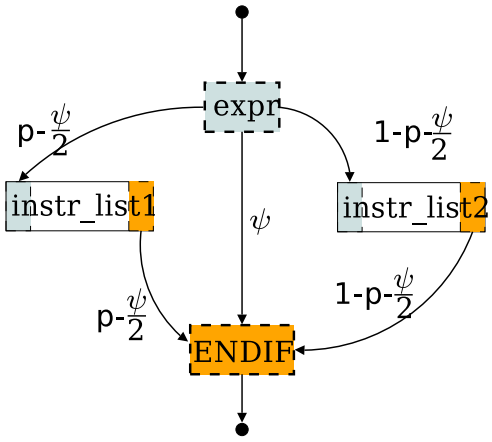

Figure 4.5: Weighted control flow for an `IF` instruction.

of the `IF` instruction. Notice that $p - \frac{\psi}{2}$ expresses the probability that the condition of the `IF` instruction evaluates to `TRUE`, hence $p \leq 1 - \frac{\psi}{2}$. The total weighted control flow induced by the `IF` instruction is its additional weighted control flow, together with the total weighted control flow induced by instr_list$_1$ multiplied by $p - \frac{\psi}{2}$ and the same for instr_list$_2$ multiplied by $1 - p - \frac{\psi}{2}$.

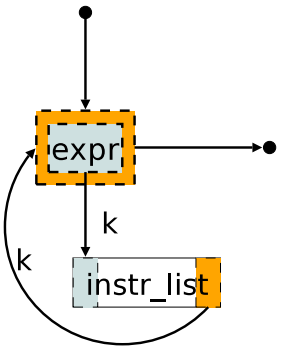

Figure 4.6: Weighted control flow for an `WHILE` instruction.

Figure 4.6 represents the case of a `WHILE` instruction, which can be derived from the rule *instr* ::= **"WHILE"** *expr* **"DO"** *instr_list* **"END_WHILE"**. The first and last part are defined as the evaluation of the condition of the `WHILE`. The additional weighted control flow introduced by the use of a `WHILE` instruction consists of weight k between the evaluation of the condition and the first instruction of the body, and likewise for the last instruction of the body. Notice that k models the expected number of times the condition of the while evaluates to `TRUE` successively. The total

weighted control flow induced by a `WHILE` is its additional weighted control flow, together with the weighted control flow induced by its body, multiplied by k.

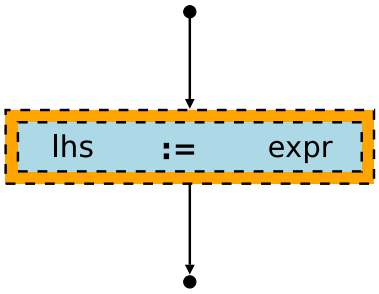

Figure 4.7: Weighted control flow for an assignment instruction.

Figure 4.7 represents the case of any simple instruction, such as assignment, method call or `WAIT` instruction. The first and last instruction for such simple instructions is the instruction itself, and no additional weighted control flow is introduced by such an instruction.

We can use this informal representation to define the expected number of exchanged messages for a given `SEQUENCE`. Indeed, given a certain `SEQUENCE`, which body is an instruction_list, we can recursively enumerate all weighted control flows. To take a certain coloring into account, we only take the weights which connect instructions of different colors.

4.3.2 Formal definition

In analogy with the formal definition of the atomic coloring problem, we define a graph G_s for sequential instructions, where vertices are linked to instructions, and an edge exists between two vertices if control may flow from one linked instruction to another. Furthermore, edges are weighted by the expected number of times control flows from one linked instruction to another.[1] In addition to these control flow edges, extra edges with infinite weight will model the fact that two instructions must be localized on the same site.

Formal definition of weighted control flow

In order to formally define the graph G_s, we first need to formalize control flow and the associated weights. We do this on a subset of the dSL grammar

[1] We suppose that there exists a factor by which all weights can be multiplied in order to obtain natural weights.

from appendix B, which is depicted in figure 4.8. This subset includes all control flow-related constructs of the dSL language. Variable, method and sequence declarations are left out because they do not involve weighted control flow.

(R1)	instruction_list	::=	instruction "**;**" instruction_list | ϵ
(R2)	instruction	::=	assign
		|	wait
		|	if_then
		|	if_else
		|	while
		|	call
(R3)	assign	::=	lhside "**:=**" rhside
(R4)	wait	::=	"**WAIT**" rhside
(R5)	if_else	::=	"**IF**" rhside "**THEN**" instruction_list "**ELSE**" instruction_list "**END_IF**"
(R6)	if_then	::=	"**IF**" rhside "**THEN**" instruction_list "**END_IF**"
(R7)	while	::=	"**WHILE**" rhside "**DO**" instruction_list "**END_WHILE**"
(R8)	call	::=	opt_launch opt_lhside ID "**(**" rhside_list "**)**"
	opt_launch	::=	"**LAUNCH**" | ϵ
	opt_lhside	::=	lhside "$<-$" | ϵ

Notice that *lhside* and *rhside* are not defined here. They are defined in appendix B, and play no role here.

Figure 4.8: Control flow sensitive subset of dSL.

As illustrated in the previous section, we need to take into account the evaluation of the conditions of `IF` and `WHILE` instructions, through vertices in G_s. Since the evaluation of an expression technically is not an instruction, we cannot define the set of vertices in G_s as a subset of the instructions in the dSL program. We therefore define basic instructions. Each basic instruction is associated to an instruction in the dSL program and is defined as a tuple (id, m, D, U) where id uniquely identifies the instruction, m identifies the method or sequence in which the instruction occurs, D is the set of defined variables in the instruction, U is the analogue for used variables.

We now define an attribute grammar [Aga76] to calculate (1) the set of basic instructions which will be vertices in G_s, and (2) the weighted

control flow between these basic instructions which will be the edges in G_s. The attribute grammar calculating these sets is defined on the subset of $_{\mathrm{d}}$SL presented earlier, in figure 4.8. We define on each of these rules the following attributes :

- b which contains the basic instruction associated to each rule.
- *first* which denotes for compound instructions the basic instruction executed first.
- *last* which denotes for compound instructions the basic instruction executed last.
- e which is the set of basic instructions induced by the rule.
- w which is the set of weighted control flows induced by the rule. w is a set of tuples (b, b', u) where b and b' are basic instructions and u is a weight.
- t which is the expected number of times control flows through the basic instruction associated to this rule.

Note that b, *first*, *last*, e and w are synthesized attributes, while t is an inherited attribute. The $.m$ attribute of basic instructions is supposed to be inherited. We now give for each of these grammar rules, the way in which these attributes are calculated.

(R1) instruction_list$_0$ ::= instruction "**;**" instruction_list$_1$:
instruction_list$_0$.b = ϵ
instruction_list$_0$.e = instruction.e $\cup$ instruction_list$_1$.e,
instruction_list$_0$.*first* = instruction.*first*,
instruction_list$_0$.*last* = instruction_list$_1$.*last*,
instruction_list$_0$.w = {(instruction.*last*, instruction_list$_1$.*first*,
instruction_list$_0$.t)} $\cup$ instruction_list$_1$.w,
instruction_list$_1$.t = instruction.t = instruction_list$_0$.t

(R2) instruction ::= assign | wait | if | while | call :
For each of these rules, b, *first*, *last*, w and e are propagated from the bottom to the top of the abstract grammar tree, while t is inherited from top to bottom. For the assign case this results in :
instruction.b = assign.b
instruction.*first* = assign.*first*
instruction.*last* = assign.*last*
instruction.w = assign.w

instruction.e = assign.e
assign.t = instruction.t

(R3) assign ::= lhside "**:=**" rhside :
assign.b = (assign.id,assign.m,Var(lhside),Var(rhside))
assign.*first* = assign.b,
assign.*last* = assign.b,
assign.e = {assign.b}
assign.w = $\emptyset$
The .t attribute is inherited, and is not propagated by this instruction.

(R4) wait ::= "**WAIT**" rhside :
wait.b = (wait.id,wait.m,$\emptyset$,Var(rhside))
wait.*first* = wait.b,
wait.*last* = wait.b,
wait.e = {wait.b}
wait.w = $\emptyset$
The .t attribute is inherited, and is not propagated by this instruction.

(R5) if_else ::= "**IF**" rhside "**THEN**" instruction_list$_1$ "**ELSE**" instruction_list$_2$ "**END_IF**"
if.b = (if.id,if.m,$\emptyset$,Var(rhside))
if.*first* = if.b
if.*last* = (endif.id,if.m,$\emptyset$,$\emptyset$) (We suppose that a unique identifier endif is available for each `IF` instruction, to denote its `END_IF` part)
if.e = {if.b, if.*last*}$\cup$ instruction_list$_1$.e $\cup$ instruction_list$_2$.e
if.w = {(if.b, instruction_list$_1$.*first*.e, if.$t \cdot (p - \frac{\psi}{2})$),
(if.b, instruction_list$_2$.*first*.e, if.$t \cdot (1 - p - \frac{\psi}{2})$), (if.$b$, if.*last*, ψ),
(instruction_list$_1$.*last*, if.*last*, if.$t \cdot (p - \frac{\psi}{2})$), (instruction_list$_2$.*last*, if.*last*, if.$t \cdot$
$(1 - p - \frac{\psi}{2})$)} $\cup$ instruction_list$_1$.w $\cup$ instruction_list$_2$.w
instruction_list$_1$.t = $(p - \frac{\psi}{2})$·if.t, instruction_list$_2$.t = $(1 - p - \frac{\psi}{2})$·if.t
Remark here that $p - \frac{\psi}{2}$ is the probability that the condition of the `IF` evaluates to `TRUE`, while $1 - p - \frac{\psi}{2}$ is the probability of the opposite. Note also that ψ is the probability that the condition evaluates to `UNKNOWN`, in which case control flows from the evaluation of the condition to the end of the `IF`. Note also that different ψ and p values can be used for each `IF` in the program text.

(R6) if_then ::= "**IF**" rhside "**THEN**" instruction_list "**END_IF**"
if.b = (if.id,if.m,$\emptyset$,Var(rhside))
if.*first* = if.b
if.*last* = (endif.id,if.m,$\emptyset$,$\emptyset$)
if.e = {if.b, if.*last*}$\cup$ instruction_list.e

if.w = {(if.b, instruction_list.*first*.e, if.$t \cdot p$) , (if.b, if.*last*, $1-p$), (instruction_list.*last*, if.*last*, if.$t \cdot p$)} $\cup$ instruction_list.w
instruction_list.t = p·if.t

(R7) while ::= "**WHILE**" rhside "**DO**" instruction_list "**END_WHILE**" :
while.b = (while.id,while.m, $\emptyset$,Var(rhside))
while.*first* = while.b
while.*last* = while.b
while.e = { while.b}$\cup$instruction_list.e
while.w = {(while.b, instruction_list.*first*, k), (instruction_list.*last*, while.b, k)} $\cup$ instruction_list.w
instruction_list.t = k·while.t Remark here that k is the estimated number of times the loop will execute. k can be dependent on the context, i.e. different values for k can be used for different loops.

(R8) call ::= opt_launch opt_lhside ID "(" rhside_list ")" :
call.b = (call.id,call.m,$\emptyset$,Var(rhside_list))
call.*first* = call.b,
call.*last* = call.b,
call.e = {call.b}
call.w = $\emptyset$
The .t attribute is inherited, and is not propagated by this instruction.

Note that with correct k, p and ψ values, the weighted control flow calculated by the .w attribute of an instruction list amounts to the average number of times control flow passes between instructions in the list. In general, calculating correct k and p values is obviously undecidable. Indeed, there is no algorithm, in general, which can decide whether a certain point in the program text is reachable or not. In practice, either profiling tools or manual input can be used to get approximate values.

The Sequential Color Graph

Each body of a method and sequence has an instruction_list, for which we can use the .e and .w attributes to construct the weighted control flow graph. The set of vertices induced by each method and sequence are in instruction_list.e, while the weighted edges are defined by instruction_list.w. Note that two instructions using/defining the same global variables, must necessarily be located on the same site (even if both instructions appear in different `SEQUENCE`s or `METHOD`s). We model this by inserting edges of infinite weight between such instructions. To complete the graph with these edges, we can use the parts D and U defined in the basic instructions.

Definition 24 (Sequential Color Graph) *For a given set of sites S, a sequential color graph is a weighted colored graph $G_s(V, E, w)$ associated to a $_d$SL program P, and a localization table T, where $V = \{b | b \in instruction\_list.e \wedge instruction\_list \in P\}$ is the set of vertices, $E \subseteq \{\{n, n'\} | n, n' \in V, \}$ is the arc set and $w : E \mapsto \mathbb{N}$ gives the weight of the edges. E and w are defined as follows :*

- *if $b = (id, m, D, U), b' = (id', m', D', U') \in V, b \neq b'$ and $(D \cup U) \cap (D' \cup U') \neq \emptyset$, then $\{b, b'\} \in E$ and $w(\{b, b'\}) = \infty$*
- *else if $(b, b', p) \in instruction\_list.w$, then $w(\{b, b'\}) = p$*

♦

The Sequential Coloring Problem

The Sequential Coloring Problem can now be formally stated as follows :

Definition 25 (The Sequential Coloring Problem (SCP)) *Given a $_d$SL program, one if its Sequential Color Graph $G_s(V, E, w)$, a localization table T. The Sequential Coloring Problem (SCP) consists in finding a coloring function $c : V \mapsto S$, compatible with T such that the sum of the edges in E between vertices of different color is minimum. i.e., such that*

$$\sum_{\{n,n'\} \in E, c(n) \neq c(n')} w(\{n, n'\})$$

is minimum. ♦

Remark *This problem can equivalently be stated as follows : Find a partition $V_1, \ldots, V_{|S|}$ of V such that*

- $\forall n, n' \in V, \{n, n'\} \in E, c(n) \neq c(n') : \nexists i \in 1..|S| : \{n, n'\} \subseteq V_i$
- $\forall n, n' \in V, \{n, n'\} \in E, c(n) = c(n') : \exists i \in 1..|S| : \{n, n'\} \subseteq V_i$

minimizing $\sum_{\{\{n,n'\} \in E | \nexists i : \{n,n'\} \subseteq V_i\}} w(\{n, n'\})$. We will use this formulation in what follows.

We show that the SCP is equivalent to the multiterminal cut problem, which is known to be an NP-Hard problem. Next, in order to obtain more general results, we focus on the multiterminal cut problem, and highlight some interesting observations, which we shall use to design efficient heuristics.

Running example

Before going into the complexity analysis of this problem, let us have a look again at our running example, the heater from figure 3.5, section 3.2.

As mentioned in the previous section, only one instruction is left uncolored after the atomic coloring problem was solved. Figure 4.9 shows the interesting part of G_s, which contains that particular instruction. Note that we chose a probability of $\frac{1}{2}$ for both branches of the `IF` (hence 0 for ψ).

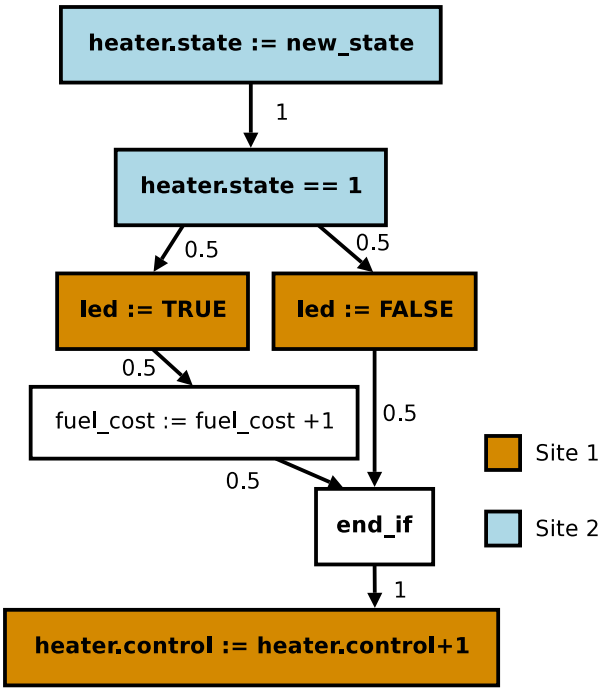

Figure 4.9: G_s of the heater example.

It is clear from the graphical representation of G_s that in this case, the instruction `fuel_cost := fuel_cost +1;` (and the `END_IF`) must be assigned to the same site as the site of `led` (site 1) in order to minimize the number of expected messages during execution.

4.3.3 Complexity results

We use the multiterminal cut problem to show the complexity of the SCP.

Definition 26 (Multiterminal Cut Problem (MCP)) *Given a weighted undirected graph* $G(V, E, w) : E \subseteq \{\{u, v\} | u, v \in V \wedge u \neq v\}$ [2], $w : E \mapsto$

[2]For technical reasons looping edges (v, v) will be omitted in all graphs considered here. Note that their presence does not change the problem.

$\mathbb{N}_0 \cup \{\infty\}$ *and a set of terminals* $T = \{s_1, ..., s_k\} \subseteq V$*, find a partition of* V *into* $V_1, ..., V_k$ *such that* $s_i \in V_i \ \forall i \in [1, k]$ *and* $\sum_{v \in V_i, v' \in V_j, i \neq j} w(v, v')$ *is minimized.* ♦

We know that the multiterminal cut problem is NP-Hard [DJP+94] for any fixed $k > 2$, even when all weights are equal to 1. When $k = 2$, the problem reduces to the minimum-st cut problem, for which a polynomial solution was first given by Ford and Fulkerson [FF62]. We now show that the multiterminal cut problem and the sequential coloring problem are equivalent.

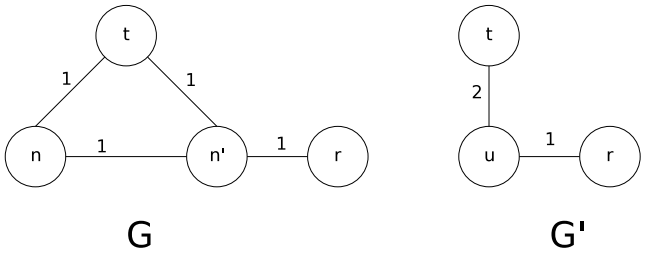

Figure 4.10: Merging operation.

In what follows, we frequently use the *merging* operation on weighted graphs. A graph G' is the result of the merging of n and n' in G, when n and n' are replaced in G' by a new vertex u and all edges $\{n'', n\}$ are replaced by $\{n'', u\}$, likewise for all edges $\{n'', n'\}$ which are replaced by $\{n'', u\}$. If this would result in multiple edges $\{n'', u\}$, then those edges are replaced by a single edge which is weighted by the sum of all those edges. The merging operation is also called *contraction* [Die05] if an edge exists between n and n', since that edge disappears from the graph. The graph G' resulting from the merging operation of n and n' in the example graph G is represented in figure 4.10. Notice that the contraction of edges with infinite weights between non terminals (in $V \setminus T$) does not influence the optimal solution for the SCP and the multiterminal cut problem.

Theorem 3 *There exists a polynomial-time reduction from the SCP to the multiterminal cut.* ■

Proof. Take the sequential color graph G_s, and merge all vertices which have edges of infinite weights. This way, we only have at most one vertex using a given variable, since vertices using the same variables have edges of infinite weight between them. Furthermore, merge all vertices together which have the same (fixed) color, in this way there is a single vertex for

each color in the localization table. Let G' be this new graph, and let $T = \{s_1, s_2, \ldots, s_k\}$ be the set of vertices of G' using an external variable. It is easy to see that G', T is an instance of the multiterminal cut and that an optimal solution to this instance is an optimal distribution of the original program. Indeed, starting from the optimal solution for the multiterminal cut $(V_1, \ldots, V_k)$, we can build an optimal partition $P_1, \ldots, P_k$ such that P_i is the set of basic instructions such that their vertices in the control flow graph is in V_i. Note that G', T can be computed from G in polynomial time since the merging operation can be implemented in $O(|E|)$ and at most $|V|$ mergings are needed. □

Remark Theorem 3 shows how an instance of the Sequential Coloring Problem can be transformed into an instance of the Multiterminal Cut Problem. In order to do so, the atomic constraints resulting in edges of infinite weight are safely contracted. In practice, this transformation is also used for the Atomic Coloring Problem : all instructions (atomic and sequential) are inserted in a graph, where atomic control flow is modeled with edges of infinite weight. Next, all edges of infinite weight are contracted and thus removed from the graph. When such an edge is contracted between two vertices of (fixed) different colors, the program is not distributable and hence rejected, since this means that there was a connected component in G_a with vertices of different color. We therefore do not consider this case.

The next lemma states that, starting from an unweighted graph, we can build a program such that the control flow graph, after merging, is equal to the original graph (except for a constant factor 2 between the weights on the edges).

Lemma 7 *Let $G(V, E)$ be an unweighted graph, then there exists a program P such that the control flow graph $G_f = (V_f, E_f, w_f)$ of P, after merging, is such that $V = V_f,\ E = E_f$, and $\forall \{v, v'\} \in E,\ w_f(\{v, v'\}) = 2$.* ■

Proof. Let $G(V, E)$. We can build a program P such that, after having merged vertices using the same variables in the control flow graph, the resulting graph G_f is equal to G. Moreover, this program contains only a list of assignments. We define the set of global variables as $X_V = \{x_v |\ v \in V\}$, i.e. there is a bijective mapping between the set V and the set of global variables X_V. We show the construction of P by induction on the size of V. The case where $V = \{v\}$ (i.e. $|V| = 1$) is obvious, we only have one instruction $i \equiv x_v \leftarrow 1$.

Suppose therefore that $|V| = n$, and let P be the corresponding program using instructions $\{x_u \leftarrow 1 | u \in V\}$. We now build a program P' for the graph $G' = (V' = V \cup \{v\}, E' = E \cup \{\{v, v_1\}, \dots \{v, v_\ell\}\}$. For this, we modify P as follows :
For all $j \in [1, \ell]$ (i.e. for all new edges), let x_{v_j} be the global variable corresponding to v_j, and let $i \equiv x_{v_j} \leftarrow 1$ be an instruction of P. In P', we replace i by : $i; x_v \leftarrow 1; i$. Note that P' is still a list of assignments. Let G'_f be the control flow graph of P', after merging, we have that G'_f contains all edges of E'. Note that all edge weights in G'_f are equal to 2. □

Note that the number of assignments of P is polynomial in the size of G because each edge in E is represented in P using 2 instructions. Since all edges in G_f have weights equal to 2, it is easy to see that G and G_f have the same optimal solution for the multiterminal cut. An illustration for this theorem is depicted in figure 4.11. An example graph G is depicted first. Next, the figure shows the addition of instructions in P and, for each step, the corresponding G_f.

Theorem 4 *There exists a polynomial-time reduction from multiterminal cut on unweighted graphs to the SCP.* ■

Proof. Let $G(V, E), T$ be an unweighted instance of multiterminal cut, by lemma 7, we can build a program P which has its control flow graph G_f after merging equal to G (except for the constant factor of 2 on the weights).

Recall that a variable in P is associated to each vertex in G. Let the number of sites in the SCP be equal to $|T|$, and let $h : T \mapsto S$ be a bijection from the set of terminals in G to the set of sites. Specify (in the localization table of the SCP) that for each terminal $t \in T$, the color of the instructions using the variable associated to t must be $h(t)$.

From theorem 3, we know that it is equivalent to solve the multiterminal cut on G_f than to solve the SCP on P. □

We know that multiterminal cut on unweighted graphs is NP-hard. Thus, we can state the following corollary.

Corollary 3 *The SCP is NP-hard, even when the program contains only a sequence of instructions.* ■

Corollary 4 *The SCP is polynomially equivalent to the multiterminal cut problem on arbitrary graphs.* ■

This results from theorems 3 and 4. Arbitrary weights $2w$ can be obtained in the construction of theorem 4 using the `while` construct as follows : replace

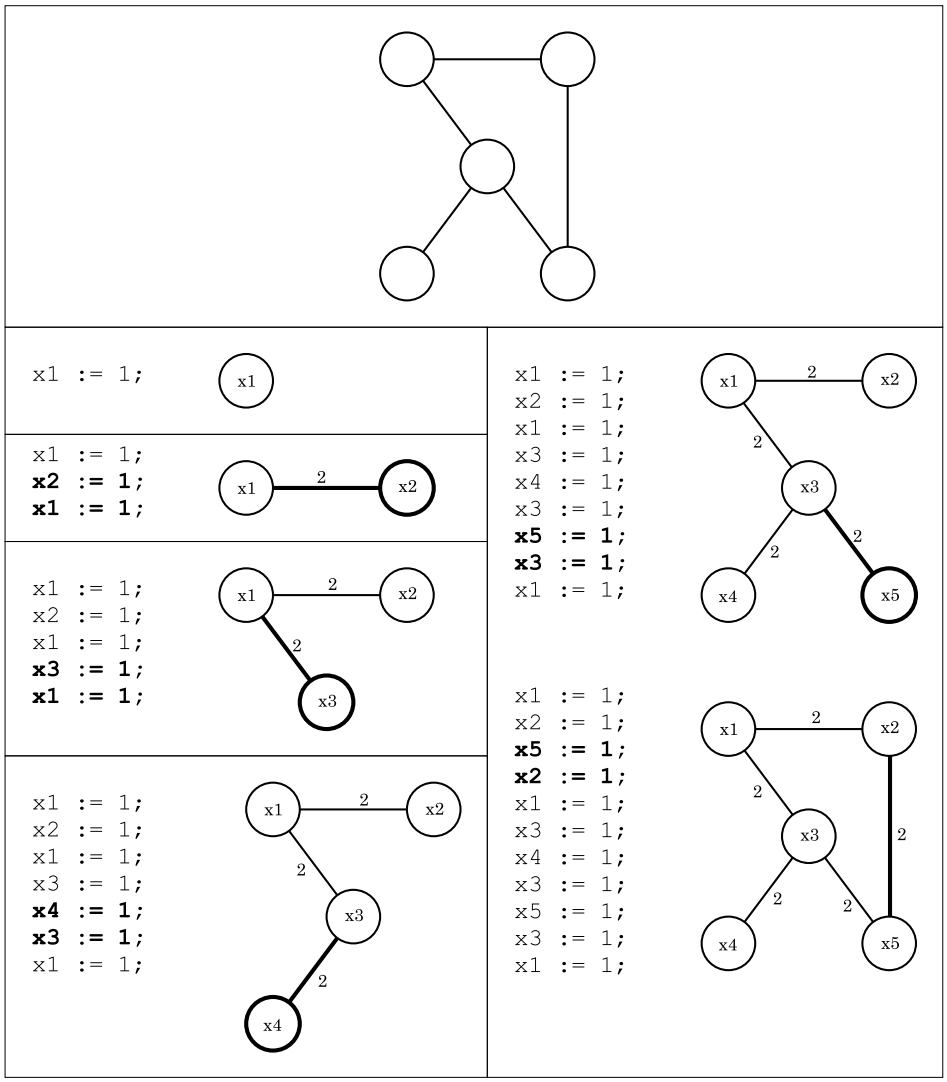

Figure 4.11: Illustration of lemma 7

the sequence $i; x_v \leftarrow 1; i$ with $i; x_v \leftarrow w;$ `while` $x_v > 1$ `) DO` $i; x_v \leftarrow x_v - 1$ `; END_WHILE;` i.

4.3.4 Related work

The multiterminal cut problem has been studied first by Dahlhaus *et al.* in [DJP+94]. In this paper, the authors prove that this problem is NP-hard for any $k > 2$ even when k is fixed where k is the number of terminals. The problem is polynomially solvable when $k = 2$, a well known result proved by Ford and Fulkerson [FF62], and in the case of planar graphs. The authors also present a $2 - \frac{2}{k}$ polynomial time approximation algorithm that relies on *isolating* cuts, a technique that is detailed further on. Moreover, they proved that this problem is MAX SNP-hard, i.e. there is no polynomial time approximation scheme unless P=NP. In [CKR00], Calinescu, Karloff, and Rabani, presented a linear programming relaxation. Using this technique and a well chosen rounding procedure, they obtain an approximation factor of $1.5 - \frac{1}{k}$. This factor was lowered to 1.3438 by Karger *et al.* in [KKS+99] who give better approximations when $k \geq 14$. These improvements were found by studying carefully the integrality gap and giving a more precise rounding procedure. A polyhedral approach [CO96, SC91, Cun91] and a non-linear formulation [DB95] have also been studied for the multiterminal cut problem.

Shrinkage, introduced in the next section, has also been studied by Högstedt and Kimelman in [HK01]. In this paper, the authors give some optimality-preserving heuristics that allow to reduce the size of the input graph by contracting some edges. The shrinkage technique presented later generalizes some of their criteria (such as independent nets and articulation points).

Here, we consider the multiterminal cut problem on undirected graphs, but work has also been done on directed graphs. Naor and Zosin presented a 2-approximation algorithm for this problem in [NZ01]. On the other hand, Costa, Letocart and Roupin proved in [CLR05] that multiterminal cuts on acyclic graphs could be computed in polynomial time using a simple flow algorithm. A generalization of multiterminal cut is *minimum multicut* where a list of pairs of terminals is given and we must find a set of edges, of minimum weight, such that these pairs of terminals are disconnected. Garg *et al.* [GVY94] give a $O(\log k)$-approximation algorithm for this minimum multicut. A survey on multiterminal cuts and its variations can be found in [CLR05].

The applications that rely on the multiterminal cut fall mainly into two

domains : the domain of parallel computation and the partitioning of distributed applications. The problems encountered in parallel computation are concerned with the allocation of tasks on different processors. The total load must be partitioned in roughly equal sized pieces, characterized by some load balancing criterion, while some interconnection criterion must be minimized ([HHLV97] and [HLLR00]). These problems can be formulated using the strongly related k-cut problem, which asks to partition the graph in k subsets such that crossing edges are minimized. Since this problem has no fixed terminals, it is polynomially solvable, for any fixed $k \geq 3$ [DJP$^+$94] and is thus considerably easier than the problem addressed here.

For the distributed applications, the problem is similar, except that it is the various application's components that must be distributed among different processors. Several criteria are studied, such as the inter object communication load of [HK01]. However, we are not aware of other work that are based on the static distribution of the instructions where the control flow is used to minimize the expected communications load. Because of this fine grain distribution, the scale of our problem is considerably larger than the studies on the partitioning of objects or functions as is the case in *classical* distributed systems. Therefore, we believe that the results of the heuristics presented here are applicable on these smaller instances as well.

4.3.5 A generalized global criterion

In [DJP$^+$94] the authors design a $2 - \frac{2}{k}$ approximation algorithm based on the isolation heuristics which uses `st` cuts. An `st` cut (multiterminal cut with $k = 2$, the terminals are called s and t) divides the vertices in the graph into two sets $(C, \overline{C})$ where $s \in C$ and $t \in \overline{C}$. The heuristics consists in finding an optimal isolating cut for each of the k terminals $\{s_1, ..., s_k\}$ and taking the union of the $k - 1$ smallest of these cuts. An optimal isolating cut is a minimum `st` cut where $s = s_i$ and t is the vertex resulting from the merging of $s_{j \neq i}$. We now introduce the original shrinkage theorem proved by Dahlhaus et al :

Theorem 5 (Shrinkage) *Given a graph $G(V, E, w)$ with terminals $T = \{s_1, ..., s_k\} \subseteq V$. Let G'_i be the graph where all terminals in $T \setminus \{s_i\}$ are merged into t, and $(C, \overline{C})$ a minimum* `st` *cut between s_i and t, then there exists an optimal multiterminal cut $(V_1, ..., V_k)$ of G such that $\exists \ell : C \subseteq V_\ell$.*
■

Theorem 5 allows us to shrink (i.e. to merge) all vertices in C into one vertex. Shrinkage is clearly an interesting way to attack the multiterminal

cut problem. Indeed, we can apply theorem 5 to all terminal vertices in order to shrink the graph. And if one can obtain a relatively small instance, then there may be hope to find the optimal solution by exhaustive search. It can also be used independently of any other algorithm designed to approximate the multiterminal cut problem. An example of the application of the shrinkage theorem is given in figure 4.12. An instance of the multiterminal cut is given in the upper left corner ($V = \{t_1, t_2, t_3, n_1n_2, n_3\}, T = \{t_1, t_2, t_3\}$). First, the theorem is applied with $s_i = t_1$. This is depicted on the right, where t_2 and t_3 are merged together, and an `st` cut is calculated between $s = t_1$ and $t = \{t_2, t_3\}$. The `st` cut is represented with a dotted line. In this example, the theorem states that there exists an optimal multiterminal cut where t_1 and n_1, n_2 are in the same partition. We can therefore merge these vertices together, and obtain the graph which is represented in the center left of the figure. The same process is repeated for $s_i = t_3$, with the resulting graph represented in the lowermost left corner. In this graph, only terminals exists, and the optimal solution is found : it consists of $\{\{t_1, n_1, n_2\}, \{t_2\}, \{t_3, n_3\}\}$ and has weight 6. Note that in this example, the shrinkage theorem gives enough reduction to find an optimal solution.

We extend theorem 5 to handle more shrinkage as follows :

Theorem 6 (More shrinkage) *Given a weighted graph $G(V, E, w)$ with terminals $T = \{s_1, ..., s_k\} \subseteq V$. Let $v \in V$ (be it in T or not), and G'_v be the graph where all terminals in $T \backslash \{v\}$ are merged into t, and $(C, \overline{C})$ a minimum* `st` *cut between v and t in G'_v then there exists an optimal multiterminal cut $(V_1, ..., V_k)$ of G such that $\exists \ell : C \subseteq V_\ell$.* ■

Proof. Let us consider a multiterminal cut $C^* = \{V_1, ..., V_k\}$ and, without any loss of generality assume $v \in V_1$. We define a partition $C^{*'}$ of V in $V'_1, ..., V'_k$ as follows :

$$V'_1 = V_1 \cup C \tag{4.1}$$

$$V'_{j \neq 1} = V_j \setminus (V_j \cap C) = V_j \setminus C = V_1 \cap \overline{C} \tag{4.2}$$

We show that $C^{*'}$ is a multiterminal cut with a weight not more than the weight of C^*, proving the existence of a multiterminal cut of the kind described in the theorem. It is easy to verify that $C^{*'}$ is a multiterminal cut, since

- $V'_i \cap V'_{j \neq i} = \emptyset$
- $\cup_i V'_i = V$

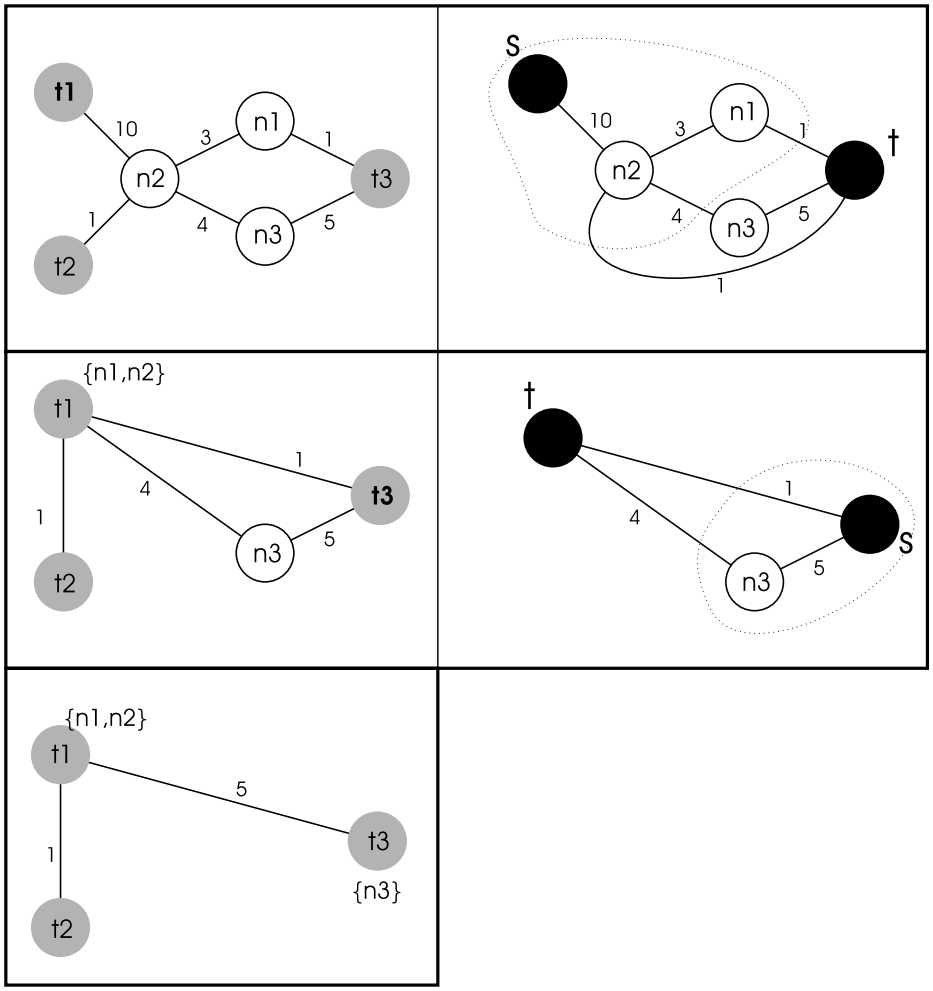

Figure 4.12: Illustration of theorem 5.

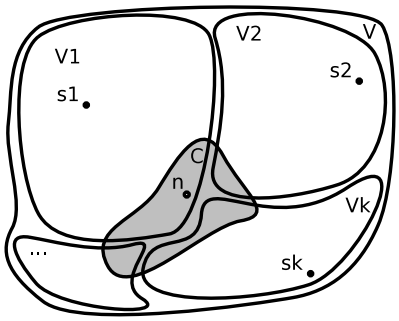

Figure 4.13: Illustration of theorem 6.

- $\forall i : s_i \in V_i'$

- $\forall i : s_i$ is isolated from $s_{j \neq i}$

Let us show that $C^{*'}$ has a weight less than (or equal to) the weight of C^*, so that if C^* is minimal, so is $C^{*'}$. Let $A, B \subseteq V$, $A \cap B = \emptyset$, we note $w(A, B) = \sum_{\{(x,y)|x \in A, y \in B\}} w(x, y)$. Furthermore, let $w(X) = w(X, \overline{X})$, where $\overline{X} = V \setminus X$. Using these notations, we have the following rules :

$$w(X, Y) = w(Y, X) \tag{4.3}$$

$$w(X, Y \cup Z) = w(X, Y) + w(X, Z) - w(X, Y \cap Z) \tag{4.4}$$

$$w(X, Y \cup Z) = w(X, Y) + w(X, Z) \quad \text{if } Y \cap Z = \emptyset \tag{4.5}$$

$$w(X, Y \setminus Z) = w(X, Y) - w(X, Y \cap Z) \tag{4.6}$$

We can naturally extend the definition of w to the evaluation of the weight of a partition of V. The weights of the two multiterminal cuts $C^{*'}$ and C^* can be expressed in terms of their partitions (resp. V_i' and V_i) as follows :

$$w(C^{*'}) = \sum_{j \neq 1} w(V_1', V_j') + \sum_{j > i > 1} w(V_i', V_j') \tag{4.7}$$

$$w(C^*) = \sum_{j \neq 1} w(V_1, V_j) + \sum_{j > i > 1} w(V_i, V_j) \tag{4.8}$$

In addition, we use two `st` cuts for proving $w(C^{*'}) \leq w(C^*)$:

$$
\begin{aligned}
w(C) &= w(C, V \setminus C) = w(C, (\cup_j V_j) \setminus C) && (4.9)\\
&= \sum_j w(C, V_j \setminus C) && \text{by (4.4), (4.5)}\\
&= w(C, V_1 \setminus C) + \sum_{j \neq 1} w(C, V_j \setminus C) && (4.10)\\
w(C \cap V_1) &= w(C \cap V_1, V \setminus (C \cap V_1)) && (4.11)\\
&= w(C \cap V_1, (\cup_j V_j) \setminus (C \cap V_1) && (4.12)\\
&= \sum_j w(C \cap V_1, V_j \setminus (C \cap V_1)) && \text{by (4.4), (4.5)}\\
&= w(C \cap V_1, V_1 \setminus C) + \sum_{j \neq 1} w(C \cap V_1, V_j) && (4.13)
\end{aligned}
$$

In order to determine $w(C^{*'}) - w(C^*)$, we will express all terms of $C^{*'}$ in terms of C^* :

$$
\begin{aligned}
w(V'_1, V'_{j\neq 1}) &= w(V_1 \cup C, V_j \setminus C) && \text{by (4.1), (4.2)}\\
&= w(V_j \setminus C, V_1) + w(V_j \setminus C, C) \\
&\quad - w(V_j \setminus C, V_1 \cap C) && \text{by (4.3), (4.4)}\\
&= w(V_1, V_j) - w(V_1, V_j \cap C) + w(V_j \setminus C, C) \\
&\quad - w(V_1 \cap C, V_j) + w(V_1 \cap C, V_j \cap C) && \text{by (4.3), (4.6)}\\
& && (4.14)\\
w(V'_{i\neq 1}, V'_{j>i}) &= w(V_i \setminus C, V_j \setminus C) && \text{by (4.2)}\\
&= w(V_i \setminus C, V_j) - w(V_i \setminus C, V_j \cap C) && \text{by (4.6)}\\
&= w(V_j, V_i) - w(V_j, V_i \cap C) \\
&\quad - w(V_j \cap C, V_i) + w(V_j \cap C, V_i \cap C) && \text{by (4.3), (4.6)}\\
& && (4.15)
\end{aligned}
$$

Using equations (4.14) and (4.15) in (4.7), we can express the difference between $C^{*'}$ and C^* using (4.8), (4.10) and (4.13) :

$w(C^{*'}) - w(C^*) =$

$$w(C) - w(C \cap V_1) \tag{4.16}$$
$$+ \sum_{j \neq 1} (-w(V_1, V_j \cap C) + w(V_1 \cap C, V_j \cap C)) \tag{4.17}$$
$$+ \sum_{i \neq 1, j > i} (-w(V_j, V_i \cap C) - w(V_j \cap C, V_i) + w(V_i \cap C, V_j \cap C)) \tag{4.18}$$
$$- w(V_1 \setminus C, C) + w(V_1 \cap C, V_1 \setminus C) \tag{4.19}$$

which proves the theorem since

- (4.16) ≤ 0 because C is a minimum `st` cut (remark that $C \cap V_1$ is a `st` cut between v and t)
- (4.17) ≤ 0 because $(V_1 \cap C) \subseteq V_1$
- (4.18) ≤ 0 because $(V_j \cap C) \subseteq V_i$ and $w(X, Y) \geq 0$
- (4.19) ≤ 0 because $(V_1 \cap C) \subseteq C$

□

Theorem 6 differs from theorem 5 because we can apply the latter only on terminal vertices, while the former can be applied to all vertices in the graph, possibly resulting in more shrinkage and therefore smaller graphs. Figure 4.14 shows a graph ($V = \{t_1, t_2, t_3, n_1, n_2, n_3\}, T = \{t_1, t_2, t_3\}$) where theorem 5 yields no reduction (it is routine to check that each `st` cut between a terminal and the merging of the other terminals results in a singleton). Theorem 6 applied on n_1 reduces the graph by 3 vertices.

We now explain how to use theorem 6 to shrink an instance of the multiterminal cut problem. Let $v \in V$, we compute the `st` cut were v is the result of the merging of all terminals in $T \setminus \{v\}$. The vertices that are in the same partition as v are merged together, with theorem 6 assuring that this preserves optimality. A chain of graphs $G_1, ..., G_l$ can therefore be calculated where each graph is the result of the optimal merging with respect to its predecessor, and where G_l cannot be reduced any further. To compute these `st` cuts, one can use the algorithm of Goldberg and Tarjan [GT88], with complexity $O(nm \log \frac{n^2}{m})$ (where $n = |V|, m = |E|$). With the results contained in the next section, we can show that when this well known algorithm is used, then $l \leq n$, resulting in a total complexity in $O(n^2 m \log \frac{n^2}{m})$.

Once a graph cannot be reduced any further, two options remain, either search exhaustively and find an optimal solution, or *unshackle* the graph. Unshackling means contracting one or more edges that likely connect vertices

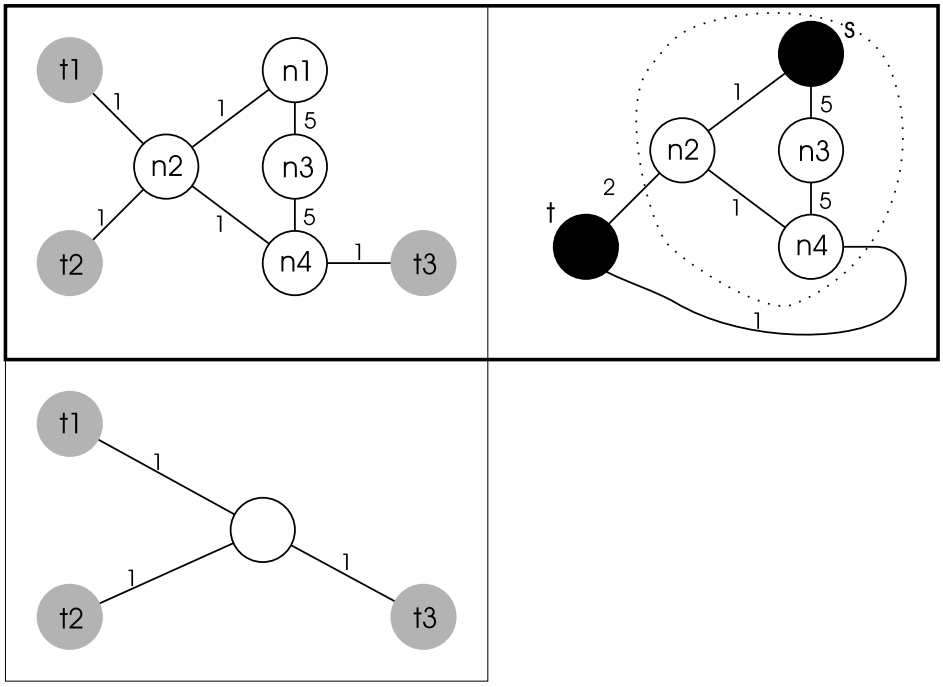

Figure 4.14: Theorem 5 and theorem 6 compared.

from the same partition in the optimal cut. Note that if an edge is picked that is in every optimal multiterminal cut, this operation will not preserve optimality. Indeed, contraction of such an edge is equivalent to forcing both vertices in the same partition of the multiterminal cut. Since none of the optimal multiterminal cuts have both vertices in the same partition, the optimal solution after contraction must be larger than the optimal solution before.

Once the graph is *unshackled*, the resulting graph may be ready for further optimal reductions. In the following section, we study an implementation using the shrinkage technique combined with a fast local unshackling heuristics.

4.3.6 A fast local heuristics

As said in the previous section, we can use the shrinkage technique in combination with an unshackling heuristics. Figure 4.15 gives a graphical overview of this technique and figure 4.16 presents an implementation. We first perform shrinkage until the graph cannot be reduced any further. Then, we use an unshackling heuristics to contract one edge from this graph. The shrinkage technique may thereupon be reused on this *unshackled* graph. This pro-

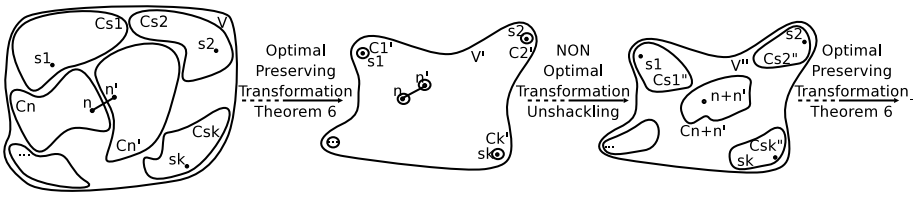

Figure 4.15: Optimal and non optimal reductions.

cess is repeated until the graph contains only terminal vertices, or is small enough to exhaustively search an optimal solution. While the resulting multiterminal cut may not be optimal, due to the unshackling heuristics, we will see that this technique generally computes a fairly good multiterminal cut and is quite efficient, provided that the unshackling is easy to compute.

```
reduce(G);
while(non-terminals exist &&
      size(G) > exhaustive_search_bound) {
    unshackle(G); // Contract 1 edge
    reduce(G);
}
```

Figure 4.16: Unshackling heuristics.

Definition and complexity

Definition 27 (MAX-MIN-st cut) *Given a weighted graph $G(V, E, w)$ and two different vertices $s, t \in V$. Define min-ST(s, t) as the set of cuts separating s and t with minimum weight. We define the set* MAX-MIN-st (s, t) *as the set of cuts $(C, \overline{C}) \in$ min-ST(s, t) such that $|C|$ is maximal.* ♦

We can easily extend these definitions for sets of vertices.

Definition 28 (MAX-MIN-st cut') *Given a weighted graph $G(V, E, w)$ and a set $T \subseteq V$,* MAX-MIN-st (s, T) *is equivalent to* MAX-MIN-st (s, t) *in the graph G where all vertices in T have been merged into the new vertex t.* ♦

We now prove some interesting properties related to those maximum size minimum cuts. Theorems 7, 8 and 9 give some remarkable insights on the structure of these cuts, which leads to a more efficient implementation of our heuristics.

Theorem 7 *Given a weighted graph $G(V, E, w)$ and two vertices $s, t \in V$, $|\text{MAX-MIN-}\texttt{st}(s,t)| = 1$, i.e. there is only one maximal size minimum* `st` *cut for any couple (s,t).* ■

Proof. By contradiction. Suppose $(S, \overline{S}), (S', \overline{S'}) \in \text{MAX-MIN-}\texttt{st}(s,t)$ ($S \neq S'$). Let $I = S \cap S'$ and $T = V \backslash (S \cup S')$, it is easy to see that S (resp. $S' \neq I$, since $S \neq S'$. As S is a min `st` cut, we have :

$$\begin{aligned}
& w(I) \geq w(S) \\
\iff\ & w(S \setminus I, I) + w(S' \setminus I, I) + w(I, T) \geq w(S, S' \setminus I) + w(S, T) \\
\iff\ & w(S \setminus I, I) + w(S' \setminus I, I) + w(I, T) \geq w(S, S' \setminus I) + w(S \setminus I, T) \\
& \qquad\qquad + w(I, T) \\
\iff\ & w(S \setminus I, I) + w(S' \setminus I, I) \geq w(S, S' \setminus I) + w(S \setminus I, T)
\end{aligned}$$

As $I \subseteq S$, we have that

$$w(S' \setminus I, I) \quad = w(I, S' \setminus I) \leq w(S, S' \setminus I)$$

Therefore, we must have

$$w(S \setminus I, I) \geq w(S \setminus I, T) \tag{4.20}$$

Now, let's compute $w(S \cup S')$, noticing that $S \cup S'$ is also a `st` cut for (s,t).

$$\begin{aligned}
w(S \cup S') &= w(S \setminus I, T) + w(S' \setminus I, T) + w(I, T) \\
&\leq w(S \setminus I, I) + \underbrace{w(S' \setminus I, T) + w(I, T)}_{= w(S', T)} && \text{by 4.20} \\
&\leq w(S \setminus I, I) + w(S', T) \\
&\leq w(S \setminus I, S') + w(S', T) && \text{since } I \subseteq S' \\
&\leq w(S')
\end{aligned}$$

Thus, $S \cup S'$ is a minimum `st` cut for (s,t). But we have $S, S' \subsetneq S \cup S'$, therefore $S, S' \notin \text{MAX-MIN-}\texttt{st}(s,t)$ and we have a contradiction. □

Since $\text{MAX-MIN-}\texttt{st}(s,t)$ contains only a single element $\{C, \overline{C}\}$ ($s \in C, t \in \overline{C}$), we can refer to $\text{MAX-MIN-}\texttt{st}(s,t)$ as *the* maximal size minimum cut. In what follows, we use $\text{MAX-MIN-}\texttt{st}(s,t)$ to designate C.

Theorem 8 *Given three different vertices $s, s', t \in V$, if $s' \in$ MAX-MIN-*`st` *(s,t), then* MAX-MIN-`st` *$(s',t) \subseteq$* MAX-MIN-`st` *(s,t).* ■

Proof. By contradiction: let $S = \text{MAX-MIN-}\texttt{st}(s,t), S' = \text{MAX-MIN-}\texttt{st}(s',t)$ and suppose that $S' \not\subseteq S$. We have that $|S \cup S'| > |S|$. Let $I = S \cap S'$ and $T = V \setminus (S \cup S')$, we define the following (cfr. figure 4.3.6 :

$$\begin{array}{lll} A \equiv w(S \setminus I, T) & B \equiv w(I, T) & C \equiv w(S' \setminus I, T) \\ D \equiv w(I, S \setminus I) & E \equiv w(I, S' \setminus I) & F \equiv w(S \setminus I, S' \setminus I) \end{array}$$

$$\Longrightarrow \quad \begin{array}{ll} w(S \cup S') = A + B + C & w(S) = A + B + E + F \\ w(S') = B + C + D + F & w(I) = B + D + E \end{array}$$

From the definition of S and S' we have $w(S \cup S') > w(S)$ (as $S \subsetneq S \cup S'$) and $w(I) \geq w(S')$, which implies that $C > E + F$ and $E \geq C + F$. This leads to a contradiction. □

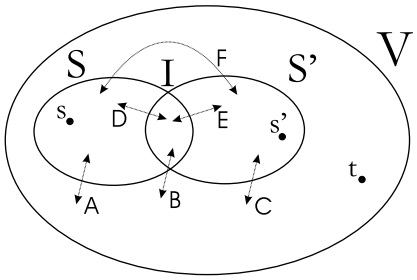

Figure 4.17: Illustration for theorem 8.

Theorem 9 *Given graph $G(V, E, w)$ and three distinct vertices $s, s', t \in V$. Let $S = \text{MAX-MIN-}\texttt{st}(s,t)$, $S' = \text{MAX-MIN-}\texttt{st}(s',t)$, $I = S \cap S'$, and $T = V \setminus (S \cup S')$. If $I \neq \emptyset$ and $S \neq I$ and $S' \neq I$, then $w(I, S \setminus I) = w(I, S' \setminus I)$. Moreover, we have that $w(I, V \setminus (S \cup S')) = 0$. The same results hold when $S = \text{MAX-MIN-}\texttt{st}(s, \{t \cup s'\})$, $S' = \text{MAX-MIN-}\texttt{st}(s',t)$ or $S = \text{MAX-MIN-}\texttt{st}(s, \{t \cup s'\})$, $S' = \text{MAX-MIN-}\texttt{st}(s', \{s \cup t\})$.* ■

Proof. By contradiction : let's reuse the equations from proof of theorem 8, to compute $w(S \setminus I)$ and $w(S)$:

$$w(S \setminus I) = A + D + F \qquad w(S) = A + B + E + F$$

As S is the MAX-MIN-`st` cut(s,t), we have $A + B + E + F \leq A + D + F$ and $B + E \leq D$. By applying a similar reasoning with S' and $S' \setminus I$, we can prove that $B + D \leq E$. In conclusion, we have $E = D (\Rightarrow w(I, S \setminus I) = w(I, S' \setminus I))$ and $B (\equiv w(I,T)) = 0$. The two other propositions are proved likewise. □

Theorems 7, 8 and 9 allow us to efficiently calculate the reduction phases of our unshackling heuristics. We know that the order in which we calculate

the cuts has no effect on the outcome of the algorithm. Moreover, we can calculate the MAX-MIN-`st` cut for a given vertex n and immediately merge all vertices on the same side of n in the cut, thus reducing the number of vertices before calculating the next MAX-MIN-`st` cut for the remaining unmodified vertices.

Theorem 10 *Let $G(V, E, w), T$ be an instance of the multiterminal cut, and let v, v' be two different vertices of G. The order in which G is reduced, relative to v and v' is irrelevant. I.e., reducing G relative to v followed by a reduction relative to v' yields the same graph as the one obtained by reducing in the opposite order.* ■

Proof. Let S be the MAX-MIN-`st` cut for s, and S' be the MAX-MIN-`st` cut for s'. We now consider all possibilities for S and S'. Figure 4.3.6 shows the different possible configurations (remark that the configuration on the right is not possible due to theorem 8).

- $S \cap S' = \emptyset$. This case is trivial, indeed, the contractions of S and S' do not interfere with each other. Thus, the order of contraction does not matter in this case.

- $S \subseteq S'$. If we first contract S, then S', we will end with a single vertex corresponding to all vertices of S'. If we first contract S', then we will directly end with a single vertex corresponding to all vertices of S'. In both cases, the edges from this single vertex will be the sum of all edges from S'. Thus, the resulting graph is the same in both orders.

- $S' \subseteq S'$. This case is symmetrical.

- $S' \cap S = I \wedge I \neq S \wedge I \neq S'$. Let's analyze the possible orders of contraction.

  If we first contract S, by theorem 8, we have the new max-min-st for s' is $S' \setminus I$. Thus, we have two vertices, n_s, the result of the contraction of S, and $n_{s'}$, the result of the contraction of $S' \setminus I$.

  On the other hand, if we first contract S', by theorem 8, we have that the new max-min-st for s is $S \setminus I$. Thus, we have two vertices, $m_{s'}$, the result of the contraction of S', and m_s, the result of the contraction of $S \setminus I$.

  Let's analyze the edges from n_s and from m_s. By theorem 8, we have that $w(I, V \setminus (S \cup S')) = 0$, thus all edges from S are edges from $S \setminus I$.

Hence, we can easily see that the edges from n_s are the same than the edges from m_s. In conclusion m_s and n_s are equivalent.

The case is symmetrical for the edges from $n_{s'}$ and from $m_{s'}$. By theorem 8, we have that $w(I, V \setminus (S \cup S')) = 0$, thus all edges from S' are edges from $S' \setminus I$. Hence, we can easily see that the edges from $n_{s'}$ are the same than the edges from $m_{s'}$. In conclusion $m_{s'}$ and $n_{s'}$ are equivalent.

□

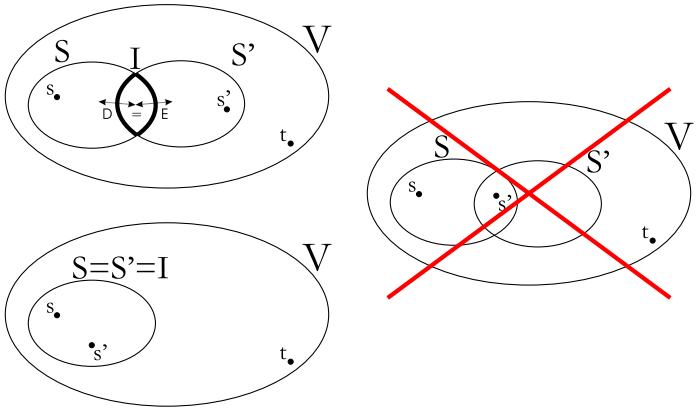

Figure 4.18: Possible configurations for MAX-MIN-st cut

After the calculation and merging of all MAX-MIN-st cuts, we have for all vertices in the reduced graph and terminals $s_1, \ldots, s_k$, MAX-MIN-st cut $(s, \cup_i s_i \setminus \{s\}) = \{s\}$. The only missing link is how to obtain the MAX-MIN-st:

Theorem 11 *The algorithm of Goldberg and Tarjan [GT88] calculating the maximum flow in $O(nm \log(\frac{n^2}{m}))$-time also yields* MAX-MIN-st ■

Proof. By contradiction. As for prerequisites, the reader is expected to be familiar with [GT88], where the authors prove that it is possible to calculate a minimum st cut $(S_g, \overline{S}_g)$ with $s \in S_g \wedge t \in \overline{S}_g$ in $O(nm \log(\frac{n^2}{m}))$-time. We will use their notations to prove that the min st cut calculated by their algorithm is in fact the unique minimum st cut of maximal size.
Let $g(v, w) : E \mapsto \mathbb{R}^+$ be the preflow function (here we may suppose that the algorithm terminated and that the preflow is a legal flow). G_g is used to indicate the residual graph and $c(v, w) : E \mapsto \mathbb{R}^+$ indicates the capacities of the edges in E. In addition, $(S_g, \overline{S}_g)$ is defined as the partition of V such that $\overline{S}_g$ contains all vertices from which t is reachable in G_g and $S_g = V \setminus \overline{S}_g$. We use the following lemma by Golberg and Tarjan from [GT88]:

When the first stage terminates, $(S_g, \overline{S}_g)$ is a cut such that every pair v, w with $v \in S_g$ and $w \in \overline{S}_g$ satisfies $g(v, w) = c(v, w)$.

Suppose that there exists another minimum cut $(C', \overline{C'})$ such that $|C'| > |S_g|$ which is maximal in size.
Remark that $S_g \subseteq C'$ because of theorem 8 and $s \in C' \cap S_g$.
Let $I = C' \cap \overline{S}_g$. Note that $I \neq \emptyset$ since $|C'| > |S_g|$. We split the boundaries between S_g, I and $\overline{S}_g$ in three sets (the situation is depicted in figure 4.19) :

- Old Boundary: $O \subseteq E = (v, w) : v \in S_g \setminus I \wedge w \in I$
- New Boundary: $N \subseteq E = (v, w) : v \in I \wedge w \in \overline{S}_g \setminus I$
- Common Boundary: $C \subseteq E = (v, w) : v \in S_g \setminus I \wedge w \in \overline{S}_g \setminus I$

By definition of $(S_g, \overline{S}_g)$, we know that $g(O) = c(O)$. We also know that since $(C', \overline{C'})$ and $(S_g, \overline{S}_g)$ are both minimum cuts : $w(O) + w(C) = w(N) + w(C) \Rightarrow w(O) = w(N)$. Remark that since g is a legal flow, the flow entering I must be equal to the flow getting out of I, which means that $g(O) = g(N)$.

The combination of these tree equations leads to a contradiction: since the edges in N are saturated, t is not reachable from any $n \in I$ in G_g which means that $I = \emptyset$. □

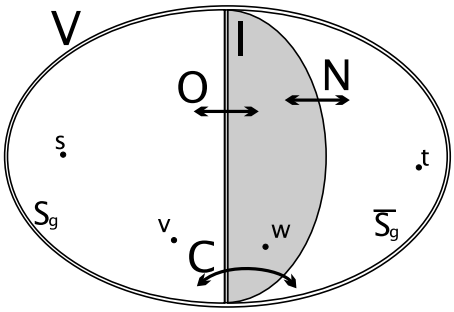

Figure 4.19: Illustration for theorem 11

Lemma 8 *Given an instance $G(V, E, w), T$ of the multiterminal cut and an edge $e = \{v_1, v_2\} \in E$. Let G' be the resulting graph after contraction of e in G. If G is irreducible (i.e. $\forall v \in V$: MAX-MIN-$\mathtt{st}(v, T) = \{v\}$), then in G', $\forall v' \in V' \setminus \{v_1, v_2\}$: MAX-MIN-$\mathtt{st}(v', T) = \{v'\}$.* ■

Proof. Proof by contradiction. We denote MAX-MIN-$\mathtt{st}$ for cuts in G, MAX-MIN-$\mathtt{st}'$ for cuts in G'. Let m' denote the vertex in G' that results from the contraction of e. Suppose that MAX-MIN-$\mathtt{st}'(v', T) \neq \{v'\}$. Let

$h : V' \mapsto 2^V$ be a mapping from the vertices from G' to G, where $\forall u' \in V' \setminus \{m'\} : h(u') = \{u\} \wedge h(m') = \{v_1, v_2\}$. We extend h to sets as follows : $h(U) = \cup_{u \in U} h(u)$. We show that $w(h(\text{MAX-MIN-}\texttt{st}'(v', T)))$ is a cut with no more weight than $w(\{v\})$, which is in contradiction with the fact that G is irreducible.

- If $m' \notin \text{MAX-MIN-}\texttt{st}'(v', T)$ then this is obvious, since only the edges to m are changed by the contraction, hence $w(h(\text{MAX-MIN-}\texttt{st}'(v', T))) = w'(\text{MAX-MIN-}\texttt{st}'(v', T))$.
- If $m' \in \text{MAX-MIN-}\texttt{st}'(v', T)$, then we have $w(h(\text{MAX-MIN-}\texttt{st}'(v', T)) \leq w(\text{MAX-MIN-}\texttt{st}'(v', T))$, since the contraction can only increase the weight of the edges connected to m.

□

Finally, we can prove that the worst execution time for the unshackling heuristics stays within the complexity of the reduction algorithm :

Theorem 12 *The unshackling algorithm from figure 4.16 can be implemented with worst case complexity* $O(n^2 m \log(\frac{n^2}{m}))$ *if the complexity of* `unshackle()` *is in* $O(nm \log(\frac{n^2}{m}))$. ■

Proof. Consider an irreducible graph in which one and only one edge $\{v_1, v_2\}$ is contracted. Before contraction, $\forall v \in V$: MAX-MIN-`st` $(v, T) = \{v\}$. By lemma 8 we know that the contraction only affects the resulting vertex from the contraction, which means that after each contraction only one MAX-MIN-`st` cut has to be calculated. Since at most n contractions are possible, the number of MAX-MIN-`st` calculations needed can be bounded by $2n$ (n for the initial reduction and n for all subsequent reductions). The worst case complexity is therefore as stated, if the complexity of `unshackle()` is $O(nm \log(\frac{n^2}{m}))$. □

Results

It remains to define the way we will unshackle the graph. We tried several local procedures, among which : Greedy, Error-reduction, and Balanced Weight. We comment each of these heuristics in more detail, and present numerical results.

Greedy The greedy heuristics takes an edge with maximal weight. This is, a priori, a good choice since it is unlikely that such an edge is in each

optimal multiterminal cut. Unfortunately, the resulting heuristics has no fixed approximation bound with this unshackling procedure.

This is illustrated in the example of figure 4.20. Consider the $6 \times n$ toroidal mesh depicted in the first part of the figure. All edges have weight 1, and terminals are represented as black vertices. For clarity reasons, some edges are not represented in the following iterations of the algorithm. Note that no reduction is possible in this graph. Since all edges have equal weight, the algorithm can pick any edge. Suppose therefore that the algorithm picks the edge marked with a thicker line. The resulting graph after unshackling is given on the top right. Note that the weights on all edges remain the same, and that the graph remains irreducible. Successive iterations of the algorithm are shown, with the end result represented in the lower right corner of the figure. The optimal solution in the original graph (upper left) consists of $\{\{t_1\}, \{t_2\}, \{t_3\} \cup V \setminus T\}$ (where $T = \{t_1, t_2, t_3\}$ is the set of terminals and V, the set of vertices) and has weight 8. The optimal solution in the graph obtained after $3 \cdot n$ iterations (lower right) consists of the analogue partition $\{\{t_1\}, \{t_2\}, \{t_3\} \cup V' \setminus T\}$ and has a weight of $4 \cdot n$. The greedy unshackling heuristics does not therefore result in an approximation algorithm since $\lim_{n\to\infty} \frac{GREEDY}{OPT} = \infty$, even for a fixed number of terminals.

Error-Reduction The error reduction unshackling procedure is based on an upper bound for the error that can be made when contracting a certain edge. The unshackling procedure picks an edge such that this upper bound is minimum. The error is expressed as the difference between the cost of the optimal solution before and after contraction.

Theorem 13 (Error-Reduction upper bound) *Given an instance $G(V, E, w), T$ of the multiterminal cut, and an edge $e = \{n, n'\} \in E$. Let $G'(V', E', w'), T$ be the result of the contraction of e in G. The difference between the cost of the optimal multiterminal cut in G' and G is bounded by $max\{0, min\{\sum_{\{n,u\}\in E\setminus e} w(\{n, u\}), \sum_{\{n',u\}\in E\setminus e} w(\{n', u\})\} - w(e)\}$, if $e \cap T = \emptyset$, and by $\sum_{\{n,u\}\in E\setminus e} w(\{n, u\}) - w(e)$ if $e \cap T = \{n'\}$.* ■

Proof. The case where $\{n, n'\} \cap T = \{n'\}$ is trivial. Consider therefore that $\{n, n'\} \cap T = \emptyset$. Let $cost_1$ be the optimal cost before reduction, $cost_2$ after reduction. For simplicity, in this proof, instead of removing a vertex from the graph G, we consider the contraction as a constraint that n and n' have to be in the same partition in the optimal solution after contraction. Let $A = \{\{n, u\} \in E \setminus e\}$ and $A' = \{\{n', u\} \in E \setminus e\}$.

(1) If there exists an optimal solution before contraction such that n and n' are in the same partition, then the reduction has no effect on the cost, and

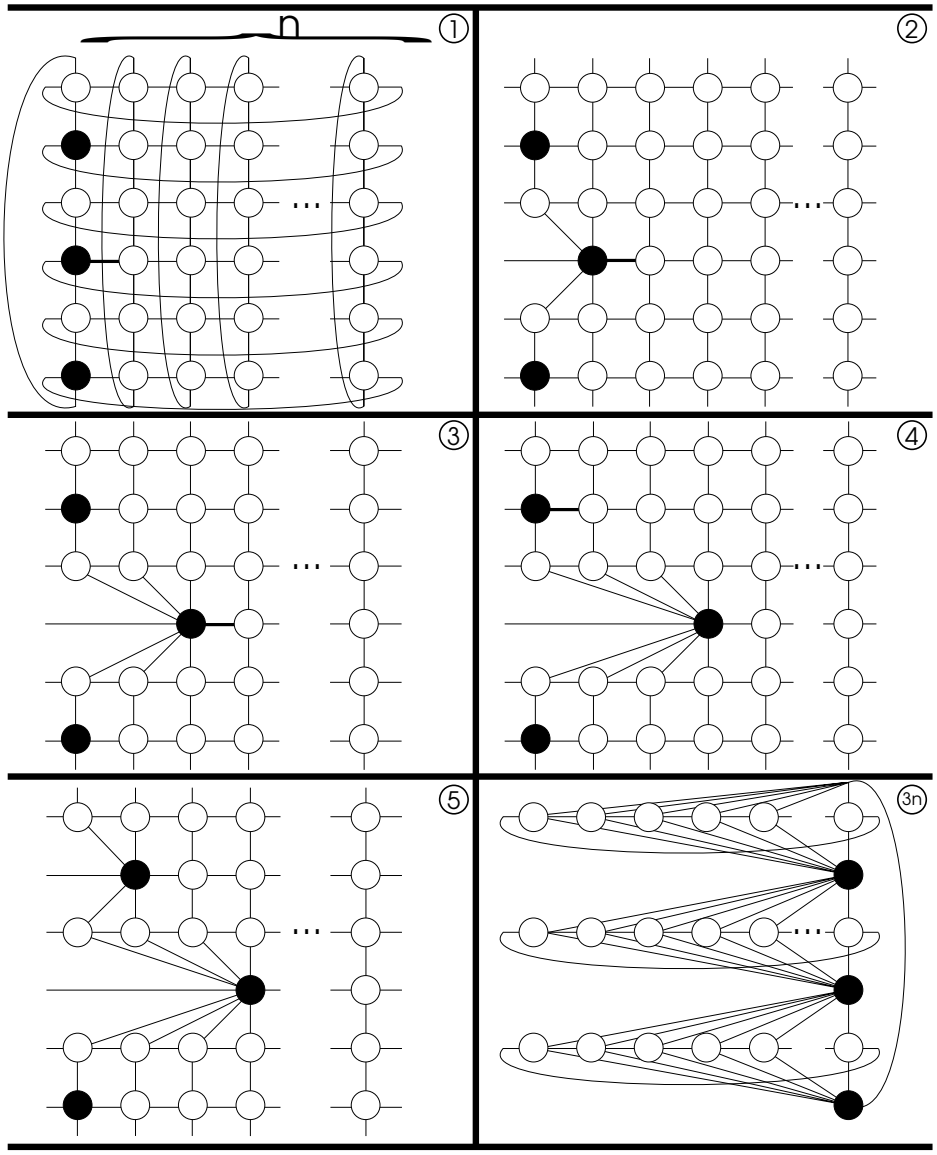

Figure 4.20: Greedy and error-reduction do not yield an approximation algorithm

$cost_1 = cost_2$.
(2) Suppose there exists no such optimal solution before contraction. Take one optimal solution $V_1, ..., V_k$ before contraction. Assume without loss of generality that $n \in V_1$ and $n' \in V_2$. Suppose $w(A) \geq w(A')$. Consider now the following multiterminal cut in G' : $V_1' = V_1 \cup \{n'\}, V_2' = V_2 \setminus \{n'\}, V_3' = V_3, ..., V_k = V_k'$. Let $cost_r$ be the cost of this multiterminal cut.
$cost_2 - cost_1 \leq cost_r - cost_1$ (since $cost_2$ is optimal).
$cost_r - cost_1 \leq w(A') - w(e)$ (by construction).
Notice that if $w(A) < w(A')$, choosing $V_1' = V_1 \setminus \{n'\}, V_2' = V_2 \cup \{n'\}, V_3' = V_3, ..., V_k = V_k'$ yields $cost_r - cost_1 \leq w(A) - w(e)$.
(1) and (2) $\Rightarrow cost_2 - cost_1 \leq max\{0, min\{w(A), w(A')\} - w(e)\}$. □

Remark *Theorem 13 provides us with a fast and local shrinkage criterion. Indeed, if an edge $\{n, n'\}$ exists such that its weight is larger than the sum of all other edges connected to n or n', then that edge can be contracted and optimality is preserved. However, it is easy to show that our shrinkage theorem (theorem 6) is more general.*

Unfortunately, again, this unshackling procedure has no fixed approximation bound. To show this, consider the previous example in figure 4.20. The error bounds on all edges are of weight 2 and we may therefore conclude that the algorithm behaves exactly as the greedy heuristics. The error-reduction unshackling procedure therefore yields no fixed approximation bound, even for a fixed number of colors. Furthermore, it behaves poorly on a large set of input graphs. The main reason for this is that the error-reduction unshackling procedure tends to pick edges for which the graph after contraction is irreducible w.r.t. theorem 6. A lot of non-optimal contractions are therefore needed before a multiterminal cut is found, which results in poor solutions.

Balanced weight Balanced weight contracts the edge $\{n, n'\}$ such that $\sum_{\{\{n,u\}\in E\}} w(n,u) + \sum_{\{\{n',u\}\in E\}} w(n',u)$ is maximal. Balanced weight picks heavy edges (since $\{n, n'\}$ is counted twice) while the result of the contraction results in a vertex with many (heavy) edges. Surprisingly, this unshackling procedure behaves best (from a quantitative point of view) on a large set of examples, but has no fixed approximation bound either.

To show that this is not the case, we use the instances depicted in figure 4.21, where $k \geq 3$ and $n \geq 1$. Each instance consists of a set T of k terminals, each of which is connected to a different non-terminal in the set A. The non-terminals in A are connected to all $(k-1)n - 1$ non-terminals

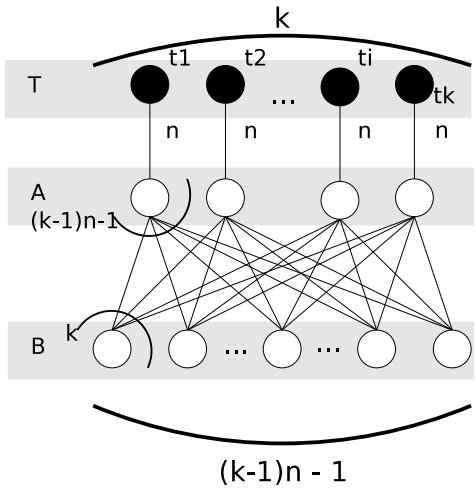

Figure 4.21: Balanced weight does not yield an approximation algorithm.

in B. In these instances the edges between vertices in T and A are weighted by n, and edges between vertices in A and B have weight 1. We first argue that all these instances are irreducible. Indeed, for all vertices t in T, it is routine to check that MAX-MIN-`st` $(t, T \setminus \{t\}) = \{t\}$. For all vertices a in A, the MAX-MIN-`st` between a and the merging of T can clearly not be a mix of vertices in A and B. It is either $\{a\}$ or $A \cup B$. In the first case, MAX-MIN-`st` has a cost of $(k-1)n - 1 + n = kn - 1$, while in the latter case it has a cost of kn. Therefore, for all $a \in A$, MAX-MIN-`st` $(a, T) = \{a\}$. For all vertices b in B, the same reasoning holds, and MAX-MIN-`st` (b, T) is either $\{b\}$ or $A \cup B$. In the first case, it is of weight k, while in the latter case, it is of weight kn. The instances are therefore irreducible for any $k \geq 3$ and $n > 1$.

The optimal solution for these instances clearly is $(k-1)n$, which is found by the Dahlhaus approximation algorithm. The heuristics value for edges between T and A is $n + (k-1)n - 1 = 2k^2 - 1$. The heuristics value for edges between A and B is $(k-1)n - 1 + k = 2k^2 - 2k - 1$. The heuristics value for the edges between T and A is larger, and are picked first by the balanced weight unshackling procedure. The resulting graph after k iterations of this heuristics is depicted in figure 4.22. Here the optimal solution is found by our algorithm, but has weight $(k-1)((k-1)n - 1)$. The solution returned by our algorithm with respect to the optimal solution for instances of figure 4.21 is therefore $\frac{\text{BALANCED}}{\text{OPT}} = \frac{(k-1)((k-1)n-1)}{(k-1)n}$, which for $k \to \infty$, and any $n > 1$ is not bounded.

Hence, unfortunately, as illustrated before, none of these heuristics has a fixed approximation bound. However, since our calculations include the

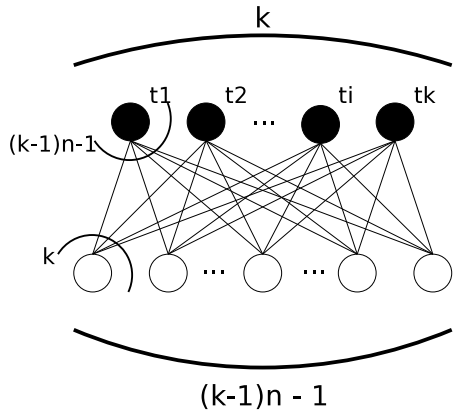

Figure 4.22: Instances after k iterations of balanced weight.

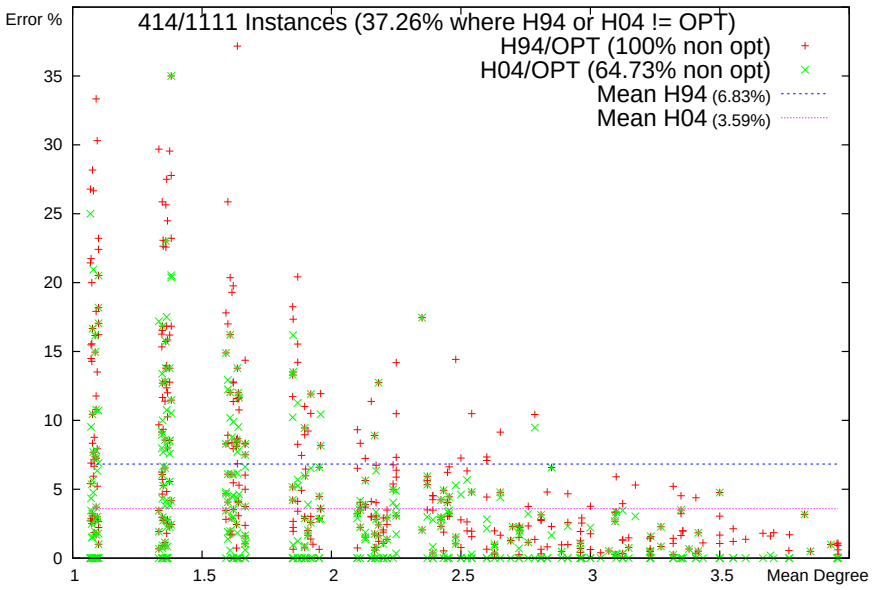

Figure 4.23: Unshackling heuristics on random graphs

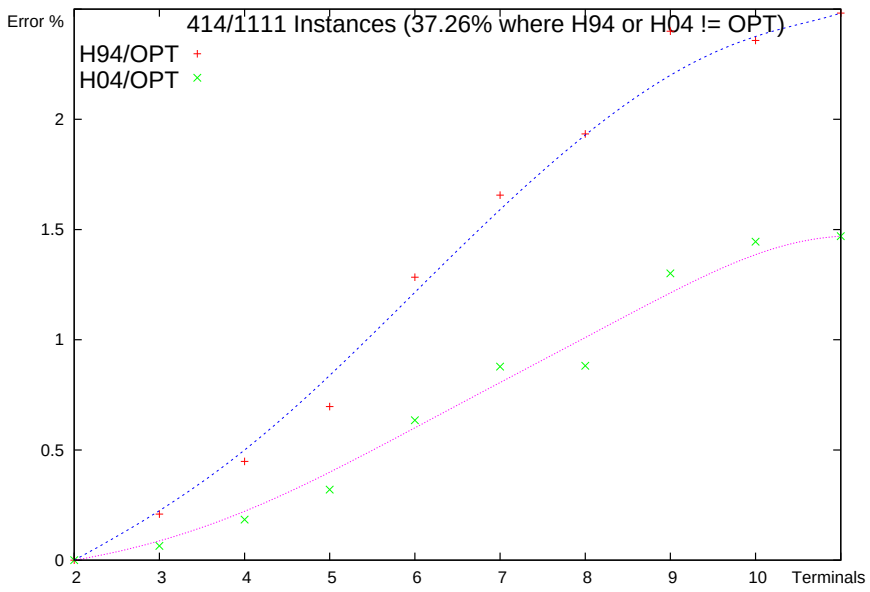

Figure 4.24: Unshackling heuristics on random graphs

ones from the $k - \frac{2}{k}$ approximation algorithm, we can compare the results and take the best of both, resulting in the same bound without any extra cost. In order to compare both heuristics, this has not been done in the following experiments.

Two sets of experiments were conducted : figures 4.23, 4.24 show the results on random graphs. These graphs are obtained by a uniform distribution of the edges between the vertices, and with weights uniformly chosen within a range that is much larger than the set of edges. Figure 4.26 shows the results on graphs obtained from auto generated programs.

In figure 4.23 we compare the results of our heuristics (indicated by $\mathcal{H}04$) with the approximation algorithm (called $\mathcal{H}94$) from Dahlhaus et al; about 1000 experiments where conducted on sufficiently small graphs (ranging from 20 to 40 vertices), allowing us to compare with the optimal solution. For 411 *hard* cases (37%), one of the heuristics failed to find the optimal. We can see that for increasing mean degree ($\frac{|E|}{|V|}$, X-axis), the error rate (Y-axis, in percent w.r.t. the optimal) for both algorithms drops rapidly, caused by the randomness in the graph. For sparse graphs however, error rates can be as high as 35%. The mean error rate for $\mathcal{H}04$, for these *hard* cases, is 3.6% while it raises to 6.8% for $\mathcal{H}94$. Remark that the failure rate for $\mathcal{H}94$ is 100% of the hard cases, while our algorithm failed in 65% of these cases. In these experiments, there was no instance where $\mathcal{H}04$ performed worse

compared to $\mathcal{H}94$.

Figure 4.24 shows the mean error rate (Y-axis, in % w.r.t. the optimal solution) for the experiments of figure 4.23, with increasing number of terminals (X-axis). We can clearly see the gain of our algorithm.

Figure 4.25 shows the mean error rate (Y-axis, in % w.r.t. the optimal solution) for the same set of instance, with increasing mean degree. Again, the benefit of our approach is clearly visible.

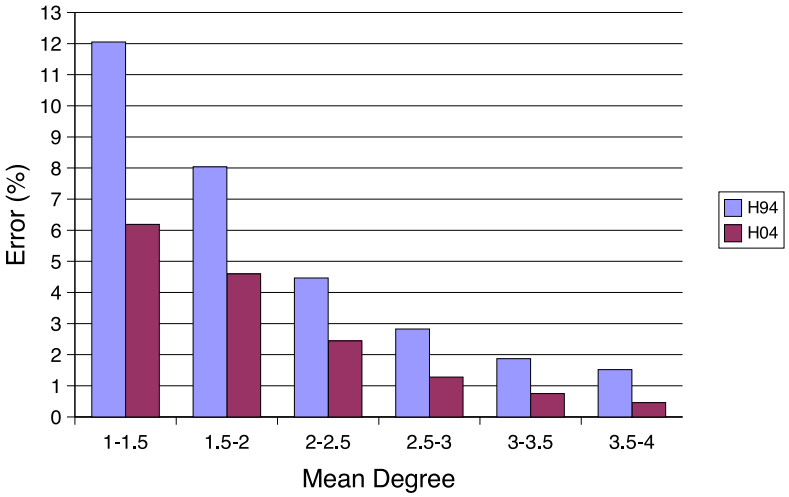

Figure 4.25: $\mathcal{H}04$ and $\mathcal{H}94$ compared on random graphs.

Figure 4.26 shows results for 25.000 grammar graphs of moderate size (600 vertices, 3 to 10 terminals), where the two algorithms are compared to each other. X-axis gives the mean degree. We can observe a difference of as high as +35% (Y-axis) for some cases, meaning that our algorithm improves the other by the same amount. For only 2 instances, our algorithm performed worse (1.3% worse and 14% worse).

4.4 Some remarks on the atomic and sequential coloring problems

Two remarks can be made on the above described problems. First, for the atomic coloring problem, for historical reasons, we defined the synchronous flow between instructions in atomic code. It can be noted that this is not needed to define the atomic coloring problem. Indeed, the structure of the code inside a `WHEN` has no influence on the atomicity of this code. It

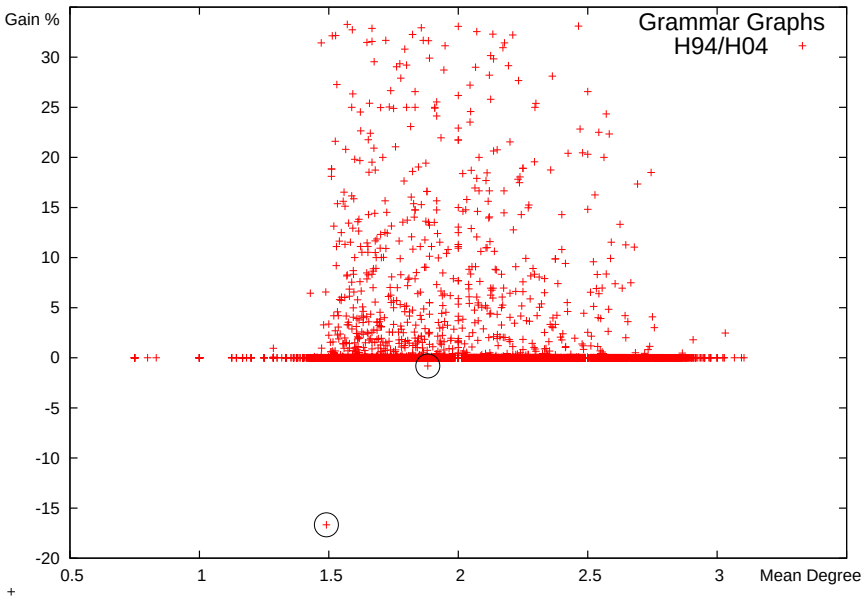

Figure 4.26: Unshackling heuristics on grammar graphs

is equivalent to model the atomic code inside a `WHEN` as a single vertex representing all instructions of its body and its condition, with edges to all (non tilded) global variables referenced by this instructions and edges to other `WHEN`s that may be triggered.

The second remark can be made in relation with the load-balancing algorithm, which is applied prior to solving the sequential coloring problem in contrast to applying it after solving the sequential coloring problem. Although this approach does not alter the distributability of a dSL program, it may lead to the production of less efficient code. To understand why, consider the instructions of a `WHEN` which is in a connected component of G_a where no vertex is colored by the localization table. Suppose now that in this `WHEN`, a reference to a global variable g is made. Clearly, g can be placed on any site, but the load balancing algorithm will fix g and all instructions involving g on a fixed site, hence restraining the possibilities for further optimizations by the sequential coloring algorithm.

4.5 Instructions reordering

In this section, we consider the problem of increasing the performance of the compiled dSL program by reordering sequential instructions. This optimization can be formulated as a well known scheduling problem, called

the precedence-constrained class sequencing problem [Tov04]. We first show how the order in which the sequential instructions are executed may influence the performance, next we formalize the problem of finding an optimal total order, and finally we give some theoretical and practical results.

4.5.1 Informal presentation

To show how the order in which the sequential instructions are executed may influence the performance of a dSL program at runtime, consider the following code, where the external variables `pump1.engine` and `pump2.engine` are localized on different sites.

```
SEQUENCE plant_startup()
...

     engine_speed1   := 10;    // (1)
     engine_speed2   := 20;    // (2)

     pump1.engine    := engine_speed1;  // (3)
     pump2.engine    := engine_speed2;  // (4)
...
END_SEQUENCE
```

Remark that the global variables `engine_speed1` and `engine_speed2` are, due to the atomicity constraint, localized on the sites of respectively `pump1.engine` and `pump2.engine`. In this case, because each instruction is localized on a different site than its successor, synchronization code will be inserted by the distributer in three points in the code : between instructions (1,2), (2,3) and (3,4). When executed, each synchronization code generates messages on the network, which has a serious impact on the performance.

The above code can be reordered in an equivalent code as follows :

```
SEQUENCE plant_startup()
...

     engine_speed1   := 10;    // (1)
     pump1.engine    := engine_speed1;  // (3)

     engine_speed2   := 20;    // (2)
     pump2.engine    := engine_speed2;  // (4)
...
END_SEQUENCE
```

In this example, only one synchronization is needed : between instructions 3 and 2. Indeed, instructions 1 and 3 are localized on the same site, which is also the case for instructions 2 and 4.

Note however, that not all instructions can be reordered. Indeed, changing the order of instructions 3 and 4 can be observed by the environment, and may lead to different behaviors, since in that case `pump2.engine` will be assigned a new value before `pump1.engine`, which is perhaps not what the programmer had in mind. Moreover, the dataflow inside a `SEQUENCE` must be respected, which means that the following two instructions may not be reordered :

```
SEQUENCE some_sequence()
...
     x := z;
     y := x + 1;
...
END_SEQUENCE
```

When cycles appear in the control flow (using `WHILE` for example), we make the hypothesis that the instructions cannot be moved.

The problem of finding an optimal reordering for an arbitrary sequence of instructions is hard. We prove this result next.

4.5.2 Formal presentation

Definition 29 (PCCS) *Given a directed acyclic graph $G(V, A)$, a set S of sites, and a surjective coloring function c from V to S. The Precedence Constrained Class Sequencing problem (PCCS) consists in finding a bijective function $f : V \mapsto \{1, .., |V|\}$ such that $f(u) < f(v)$ whenever $(u, v) \in A$ and such that $\sum_{i=2}^{|V|} \Delta(f^{-1}(i), f^{-1}(i-1))$ is minimum, where $\Delta(u, v) = 1$ when $c(u) \neq c(v)$, and 0 otherwise.* ♦

The PCCS problem appeared earlier in the Operations Research literature under the name *station routing problem* [LMT91, Lof86] and in the computer science literature as the *loop fusion problem* [Dar99]. Typically, the nodes model a set of tasks that must be performed by a number of machines which are represented by the colors. The arcs express some precedence constraints on those tasks. The PCCS consists in finding a total order in which the tasks are handled, minimizing the switchings between the different machines.

In our case, nodes represent instructions, and the arcs express the constraints which must be respected to obtain equivalent programs. These

constraints are twofold : on the one hand data dependencies must be preserved, and on the other hand access to external variables must happen in the order specified by the program text. The colors model the different sites on which the instructions must be executed.

Remark that the problem of ordering instructions is not orthogonal to the problem of coloring instructions. Here, we consider only the case where all instructions have received a color, and only reordering is allowed.

Lofgren, McGinnis and Tovey proved that PCCS is an NP-Complete problem [LMT91]. Darte additionally proved that the problem is still NP-complete even if the number of colors is a constant greater than or equal to three [Dar99]. In a more recent work, the authors of [CFSM] show that no approximation algorithm for PCCS with a constant guarantee exists, unless P=NP. Furthermore, the authors give a $\frac{|S|+1}{2}$-approximation algorithm for PCCS.

In the next section, we give an elegant proof which shows the complexity of PCCS, and give an overview of existing heuristics for PCCS. We also give a new set of heuristics, which perform well in practice.

4.5.3 Results

Complexity

The original proof that PCCS is NP-complete for $|S| \geq 3$ can be found in [Dar99]. In that paper, the author adapts a 10 pages long proof from [RU81], which is based on a reduction from Vertex Cover to Shortest Common Supersequence. We now give a shorter proof for the NP-completeness of PCCS.

To prove that PCCS is NP-complete, we focus on a restricted version of Shortest Common Supersequence (SCS) [GJ90] and show that it is a special case of PCCS.

Definition 30 (SCS) *Given a finite alphabet Σ, a finite set R of strings from Σ^* and an integer k. The Shortest Common Supersequence (SCS) asks for the existence of a string $w \in \Sigma^*$ with $|w| = k$, such that each string $x \in R$ is a subsequence of w, i.e. $w = w_1 x_1 w_2 x_2 w_2 \dots x_{|x|} w_{|x|}$, where each $w_i \in \Sigma^*$ and $x = x_1 x_2 \dots x_{|x|}$.* ♦

SCS has been proved NP-complete even on a binary alphabet $\{0, 1\}$ [RU81]. A restricted version of the SCS may be defined by requiring that no symbol of the alphabet may occur consecutively in any input string. We will refer to this as SCS-r. In [LMT91], the authors showed that SCS-r

is NP-complete on an alphabet of $2k$ symbols, if the general SCS (with consecutive repeated symbols allowed) is NP-complete for k symbols. So, with the result in [RU81] we can deduce that SCS-r is NP-complete for 4 symbols. More precise results on SCS-r are given in [Dar99], who proves its NP-completeness with only 3 symbols (note that, with 2 symbols, SCS-r is easy, and trivial with a single symbol).

Theorem 14 *PCCS is NP-complete* ■

Proof. It is clear that the problem is in NP, since the correctness of any candidate solution may be verified in polynomial time. To prove the NP-completeness, we reduce SCS-r to PCCS. Let (Σ, R, k) be an instance of SCS-r. We construct an instance of PCCS as follows.

First, we consider each symbol of Σ as a color for the vertices of the constructed graph G ($|S| = |\Sigma|$). For each $x \in R$, we form a path of length $|x|$, with vertices colored by the corresponding symbol in x. The disjoint union of all these paths is the graph $G = (V, E)$. Hence, (G, Σ) is an instance of PCCS. Notice that this construction can be done in polynomial (more precisely linear) time.

$\boxed{\Leftarrow}$ If there is a solution of PCCS, i.e., a vertex labeling of cost at most $k-1$, then there exists a string of length at most k as a solution of SCS-r. In fact, a solution of PCCS can be interpreted as a sequence of at most k blocks of vertices of V, that have the same color. If such a solution exists, then it is possible to construct a string w by replacing each block with its color. w is clearly a solution of SCS-r. Since there is no consecutive repeated color in any of the paths of G, the vertices in each block of the solution of PCCS belong necessarily to different paths. Consequently, by replacing each block with a single symbol, each $x \in R$ will have all its symbols in w. In addition, the solution of PCCS preserves the precedence constraints of G. So for each x, w can be written as $w_1 x_1 w_2 \cdots x_{|x|} w_{|x|}$, which means that w is a common supersequence.

$\boxed{\Rightarrow}$ If there is a string w of length k as a solution of SCS-r, then there is a solution of PCCS with cost at most $k-1$.

We construct k disjoint subsets $B_1, B_2, ..., B_k$ of V as follows. Define B_1 as the set of roots of G with the same color as w_1, $B_{i>1}$ as the set of vertices of $V \setminus (B_1 \cup B_2 \cup ... \cup B_{i-1})$ that have their predecessors in $B_1 \cup B_2 \cup ... \cup B_{i-1}$ and are of the same color as w_i. In order to show that $B_1, B_2, ..., B_k$ leads to a solution of PCCS with cost $k-1$, we only have to prove that $\forall v \in V, \exists i : v \in B_i$, because the construction of the B_i's preserves the precedence constraints of the vertices of G.

It is easy to see that each $x = x_1 x_2 \dots x_n$ can be written as $x = w_{i_1} w_{i_2} \cdots w_{i_n}$. such that $0 < i_1 < i_2 < ... < i_n \leq k$. Let us call P the path in G and $v_1, v_2, ..., v_n$ the vertices of P corresponding to $x_1, x_2, ..., x_n$ respectively. We proceed by induction to prove that each v_j is in B_{i_j}. By construction, $v_1 \in B_{i_1}$. Suppose now that $v_1, v_2, ..., v_{j-1}$ are in $B_{i_1}, B_{i_2}, ..., B_{i_{j-1}}$ respectively. It is clear that $v_j \notin B_1 \cup B_2 \cup ... \cup B_{i_{j-1}}$, so $v_j \in V \setminus (B_1 \cup B_2 \cup ... \cup B_{i_{j-1}})$. By hypothesis, v_j's predecessor $v_{j-1} \in B_{i_{j-1}}$, and by definition, v_j has the same color as w_{i_j}. Hence, $v_j \in B_{i_j}$. □

Heuristics

There exist several known heuristics [LMT91, KM94] for the Station Routing and the Loop Fusion problems, that may be easily adapted to PCCS. Below, we give a brief description of the most promising heuristics; but let us first consider some definitions and notations.

- A vertex with no predecessor (successor, resp.) is defined as a *root* (leaf, resp.).
- *ready* vertices of a given color a are defined as the vertices of color a which only have predecessors of color a.
- G_a denotes a graph that is obtained by removing from G the ready vertices of color a.

The idea in all the heuristics that we review in this section is to form a solution starting from the empty sequence. Then we append all the ready nodes of a chosen, with respect to some criteria, color c to the solution replacing G with G_c, and repeat the same process with the new roots until the graph is completely exhausted.

1. Greedy: At each round, choose the color c for which the number of roots is the largest. The tie is broken at random.
2. Altruist: At each round, choose the color c for which G_c has the largest number of roots. The tie is broken at random.
3. Level strategy (LS): At each round, choose the color c for which there exists a vertex in the highest level, where the ith level is defined as the set of vertices v such that the length of the longest path from v to a leaf is i. If there is more than one color in the highest level, the tie is broken at random.

4. Critical path level strategy (CPLS): Same as Level strategy, except that in the definition of levels, consecutive vertices of the same color in a path are counted for one.

5. Greedy-Altruist: Apply Greedy, but break ties by the Altruist criterion and break any remaining ties at random.

6. LS-Greedy: Apply LS, but break ties by the Greedy criterion and break any remaining ties at random.

7. LS-Altruist: Apply LS, but break ties by the Altruist criterion and break any remaining ties at random.

8. CPLS-Greedy: Apply CPLS, but break ties by the Greedy criterion and break any remaining ties at random.

9. CPLS-Altruist: Apply CPLS, but break ties by the Altruist criterion and break any remaining ties at random.

Before going into discussions, let us illustrate these heuristics by an example. Figure 1 shows a small and low-density graph related to a task labeling problem and table 1 gives a solution provided by each heuristic.

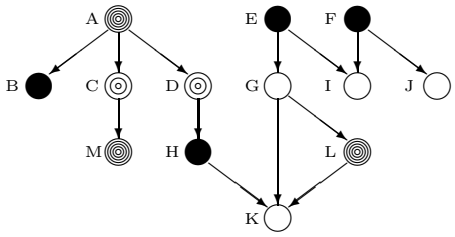

Figure 4.27: Example instance of PCCS.

The above heuristics run quickly in general, but often fail to provide "good" solutions in practice. The common weakness in all of them is, in effect, that their selections at each round are based either on local observations, or in the best case, on a very limited inspection over the respective subgraphs of the current roots:

1. The Greedy strategy just focuses on the number of occurrences of each color in the set of roots in the current state of the graph. This clearly gives no guarantee that the chosen color at each round may lead to a potentially better solution than if another color was selected.

Heuristics	*Example Solution*	*Min Cost*	*Max Cost*
Greedy	EF GIJ AL CD BH M K	6	6
Altruist	A BEF CD M H GIJ L K	6	7
Greed-Altr	Same as Greedy	7	7
LS-Greed	EF A GIJ CD BH LM K	6	6
LS-Altr	A BEF CD GIJ H LM K	6	6
CPLS-Greed	Same as LS-Greed	7	7
CPLS-Altr	Same as LS-Altr	6	6
Optimal	A CD BEFH GIJ LM K	5	

Figure 4.28: Solutions provided by the heuristics.

2. The Altruist strategy looks further in the graph but, its selection at each round is just based on a superficial "look ahead". In addition, it focuses on all the roots in G_c and not those of different colors (see the remark below). Our tests show that this "myopic" look ahead only has a little influence on the final solutions.

3. The Level strategy is based on a conceptually better criterion, as color selections someway result from some investigations in the subgraph accessible from each vertex. However just the longest paths are taken into account and not the entire subgraph accessible from each vertex. In addition (and more importantly), it only considers the vertices in the highest level, while there may be many roots of the same color in some lower levels, whose sum of the longest paths altogether is greater than that of a single root in the highest level.

4. The Critical path level strategy suffers from the same problems as Level strategy, except that its levels are defined more accurately.

Remark: As in the definition of levels in CPLS consecutive vertices of the same color in a path are counted for one, it would be nice to use this concept in the Altruist criterion as well. So, at each round, we may choose the color c for which G_c has the largest number of roots of different colors. We call this variant accordingly "Critical Altruist". Note that, the Critical Altruist heuristic would find the optimal solution of the graph of figure 4.27.

New heuristics

In our heuristics, we focus on significantly stronger parameters, namely the entire subgraph accessible from each vertex. In particular, we take into

account the "weight" of these subgraphs that may be defined with respect to various criteria.

Some notations :

- SUB(v) denotes the subgraph accessible from the vertex v.
- $W(v)$ denotes the static weight of SUB(v) (see below for various definitions).
- $W_G(c)$ denotes the weight of the color c with respect to the current state of the graph G, that is the sum of the weights of all the present roots of G that are of color c.

The general framework of all our heuristics is the same, and is essentially similar to that of the heuristics described above. What distinguishes between the new and previously known heuristics (and also among the new heuristics) is the weight definitions for the vertices and consequently for the colors.

The general idea is to select at each round a color whose weight, with respect to the current state of the graph, is maximal and to append all the roots of that color to the solution. This would make possible that an "important" portion of the graph becomes "ready" for the next round and so would give some assurance that other roots with their respective accessible subgraphs may most probably be "consumed" when processing later that subgraph. Here is the framework of our heuristics:

PCCSi **(G)**
$T :=$ empty string;
$S :=$ {colors of the roots of G};
WHILE S is not empty DO
  Set $S' = \{c \in S : W_G(c) = \max\{W_G(b) : b \in S\}\}$;
  Choose $a \in S' :$ # roots of color $a = \min\{$# roots of color $b : b \in S'\}$;
  $G := G_a$;
  $T := T \parallel$ {all newly removed vertices from G} (all of color a);
  Update S by removing a and by adding the colors of the new roots;
END
return (T);

Note that to break ties (in S'), we choose a color with minimal number of roots. In fact, since some roots of the same color may have subgraphs with some portions of nodes in common, the weight of a color may be overestimated. So, the computed weight of a color with minimal number of roots

may potentially be more close to the reality. This is actually valid for all weight definitions below.

As a first attempt, we may merely define $W(v)$ as the number of vertices in SUB(v) (that may be determined in a preprocessing phase) and the weight $W_G(c)$ of a color as the sum of the weights of the roots of that color (that should clearly be calculated at run-time). One may expect that the solution based on this criterion is generally as "good" as those returned by the Greedy, the Altruist or the LS heuristics, since all the related criteria are taken into account in the calculation of $W_G(c)$. However, we believe that this is too weak to attain sufficiently "good" solutions. For example, if a root v has a large SUB(v) but with all vertices of the same color, then it would not be possible to "consume" other roots with their respective accessible subgraphs when processing SUB(v). Thus, a realistic improvement would be to count all the roots of a same color for one, as was the motivation of the CPLS.

In the following, we consider three different criteria that may be combined in various ways to define the weights.

Criterion 1. We define $W1(v)$ as an approximation of the minimal number of sets of the vertices in SUB(v) that are of the same color and would become root at the same time. Unfortunately, the exact value of this number cannot be determined in polynomial time. However, the algorithm of figure 4.29 yields a good approximation.

```
W1 (v)
    H := SUB(v);
    i := −1;
    REPEAT
        i := i + 1;
        a := color by the Crit-Altruist criterion;
        H := H_a;
    UNTIL There is no more root in H
 return (i);
```

Figure 4.29: Criterion 1.

Observe that we select a's color at each round based on the Critical Altruist criterion. This would make it more probable that at a later stage more roots may be taken together in a set.

Criterion 2. Although defining the weights based on criterion 1 is more realistic than just the number of vertices in a subgraph, it is still not accurate. For instance, if SUB(v) consists of an alternation of two colors, then whatever W1(v), it would not be possible to "consume" the rest of the graph (that may be composed of many colors) when processing SUB(v). So, another criterion is to take into account the total number of colors in a subgraph, which is depicted in figure 4.30.

W2 (v)
H := SUB(v);
$R_{1,\ldots,m} = (0, \ldots, 0)$;
(R represents the set of colors)
REPEAT
a := random color among the current roots;
$R_a := 1$;
$H := H_a$;
UNTIL There is no more root in H
$k := \sum_{j=1}^{m} R_j$; (the number of colors in SUB(v))
return (k);

Figure 4.30: Criterion 2.

Criterion 3. Another problem with both criteria above is that the internal structure of the subgraphs is not taken into account when computing the weights. For example, if SUB(v) is just a chain with many vertices of various colors and SUB(u) consists of a graph with a vertex and a small number of direct sons, then both W1(v) and W2(v) would choose v as the root to be appended to the solution. However, SUB(u) would generally allow the rest of the graph to be "consumed" within it more "easily" than SUB(v).

This is illustrated in figure 4.31. The subgraph rooted by A can entirely be consumed when processing SUB(C), while it is not the case with SUB(B), even though both subgraphs have the same number of colors.

Another improvement is thus to assign a third weight to each vertex based on the structure of SUB(v):

When we combine the 3 criteria in the same algorithm, we obtain the algorithm in figure 4.33.

Now we can combine the 3 values i, k and z in various ways to define the weights of the vertices. We only consider weights that are the products of any subset of these criteria, i.e. we take them into account similarly and with the same importance. However, one may define $W(v)$ using other

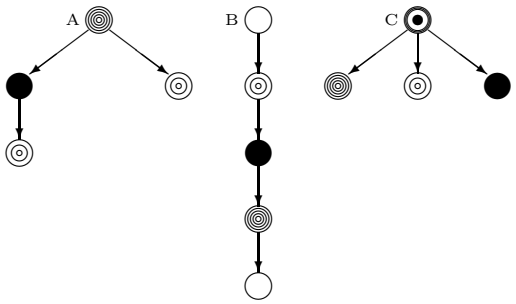

Figure 4.31: Motivation for criterion 3.

W3 (v)
$H :=$ SUB(v);
$z := 0$;
REPEAT
$a :=$ random color among current roots;
$H := H_a$;
$z := z+$ number of colors for unmarked roots;
mark new roots;
UNTIL There is no more root in H
return (z);

Figure 4.32: Criterion 3.

W (v)
$H :=$ SUB(v);
$i := -1$;
$R_{1,\ldots,m} = (0, \ldots, 0)$;
$z := 0$;
REPEAT
$i := i + 1$;
$a :=$ color from the Critical Altruist criterion;
$r_a := 1$;
$H := H_a$;
$z := z+$ number of colors for unmarked roots;
mark all the new roots;
UNTIL There is no more root in H
$k := \sum r_j$;
return (i, k, z);

Figure 4.33: Criterion 1, 2 and 3 combined.

operations as $i \times \sqrt{z^3}$, or $i \times k^2 \times \sqrt{z}$.

Results

Theorem 15 *The Greedy, Altruist, Level Strategy (and variants) and Critical Path Level Strategy (and variants) heuristics do not have a fixed approximation bound in the general case.* ■

Proof. It is easy to check that none of the mentioned algorithms guarantee to find the optimal solution for the instances depicted in figure 4.34. Note that the optimal solution consists of only n blocks $(n, n-1, \ldots, 1)$, while any of the mentioned algorithms may return a solution that consists of $\frac{n(n-1)}{2}$ blocks $(1, 2, \ldots, n, 1, 2, \ldots n-1, \ldots)$. Note that our new heuristics finds the optimal solution for these instances. Indeed, criterion 2 and 3 will pick color n first since the number of colors and roots is larger for color n. The same reasoning applies for $n-1, \ldots, 1$ □

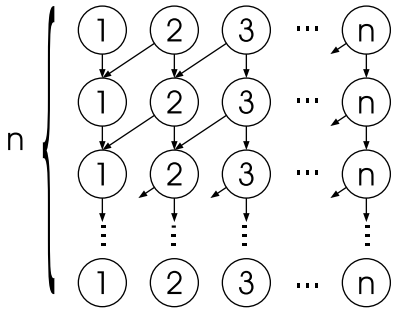

Figure 4.34: Hard PCCS instances for existing heuristics.

Theorem 16 *Level Strategy has a $|C|$-approximation bound in the case where G consists of a disjoint set of sequences.* ■

Proof. Suppose that the solution found by LS consists of k blocks of colors $c_1, ..., c_k$. The corresponding sequence of graphs is $G^0, ..., G^k$, where $G^0 = G, G^i = G^{i-1}_{c_i}$. Let w_i be the highest level in G^i. First note that $w_k = 0$, since no more nodes exist in G^k. Next, note that the sequence $w_0, ..., w_k$ is a decreasing sequence. Indeed, each time a root is removed from the graph, the highest level is either decreased or remains the same. In the latter case, roots of other colors in the same level remain in the graph. However, there can only be $|C|$ roots of different color in the same level at any given time. Thus, $k \leq |C|w_0$. Let OPT denote the number of blocks in the optimal solution. It is clear that $OPT \geq w_0$. This means that $k \leq |C|OPT$, or

otherwise stated $\frac{k}{OPT} \leq |C|$, which proves the announced approximation bound. □

For a quantitative comparison between the different existing heuristics and our new heuristics which uses $i * k * z$ as selection criterion, we used two sets of directed acyclic graphs. The first set consists of permutation graphs [BFR71]. In these graphs, each vertex corresponds to an element of the ground set of a permutation, and two vertices are adjacent if and only if the permutation reverses the relative order of the two corresponding elements. The results on permutation graphs are comparable to those obtained with interval graphs [Haj57]. That is, each vertex corresponds to an associated interval, and two vertices are adjacent if and only if the corresponding intervals intersect. The results for this first set of graphs are presented in figure 4.35. The rows in the table give the results for each heuristics, while columns indicate the number of colors in the graph. The different heuristics are compared relative to the best performing heuristics. Note that each column represents the mean result of 200 experiences, where each experience consists of running all heuristics on a permutation graphs counting 200 nodes. For example, let us take a closer look at the results using 10 colors : CPLS-Altr gets the best results, while our heuristics scores 1.1% higher, the worst heuristics (Altruist) returns solutions that are on average 8% larger. For this set of graphs, we can conclude that most of the heuristics return comparable solutions. In our opinion, this has to do with the homogeneous structure of these graphs, caused by the high amount of randomness.

Colors	2	3	4	5	10	20	30
Altruist	1	1.029	1.082	1.077	1.083	1.086	1.074
Crit. Alt.	1	1.026	1.049	1.042	1.066	1.060	1.062
LS	1	1.031	1.075	1.054	1.061	1.063	1.062
LS-Altr	1	1.032	1.064	1.045	1.030	1.032	1.034
CPLS-Altr	1	**1**	**1**	**1**	**1**	1.017	1.022
PCCS-$i * k * z$	1	1.013	1.029	1.010	1.011	**1**	**1**

Figure 4.35: PCCS heuristics on permutation graphs.

The second set of graphs are randomly created, but are constructed in such a way that they are not subject to total homogeneity. Each structured random graph in this set is constructed by using a set of trees, each having a variable width and height, and a certain number of colors. To obtain a random dag, nodes in the different trees are randomly connected (a topological sort assures that the obtained graph is a directed acyclic graph). The

results for this set of graphs are presented in figure 4.36. Each rows gives the result for the mentioned heuristics, while the columns indicate the amount of random arcs that where added between the different trees. Each column gives the mean value for 200 experiments, each of which consists of an evaluation of all heuristics on a structured random graph counting 500 nodes. Results are given for graphs counting 5 and 30 colors, and are relative to the best performing heuristics. It is clear from these figures that our heuristics does not perform well on graphs with a low number of colors, and that the performance drops when more and more random arcs are added between the trees. Remark also that the Level Strategy heuristics outperforms the altruist heuristics with solutions up to almost 40% better. To see the impact of the number of colors and the random arcs in the graph, consider figure 4.37. Here, we plotted the performance of our heuristics compared to Critical Path Level Strategy (which is the best performing existing heuristics). In *easy* cases, our heuristics comes up with solutions that are up to 8% better, while for more complex graphs, it is beaten by the Critical Path Level Strategy heuristics with roughly the same amount. The best results are obtained in graphs with a high number of colors.

Random%	20	50	100	150	200
5 colors					
Altruist	1.210	1.23	1.30	1.29	1.26
Crit. Alt.	1.375	1.33	1.30	1.26	1.24
LS	1.081	1.08	1.09	1.07	1.07
LS-Altr	1.064	1.07	1.08	1.07	1.07
CPLS-Altr	1.002	**1**	**1**	**1**	**1**
PCCS-$i * k * z$	**1**	1.01	1.04	1.0	1.08
30 colors					
Altruist	1.337	1.39	1.38	1.34	1.33
Crit. Alt.	1.523	1.47	1.38	1.31	1.27
LS	1.105	1.09	1.06	1.04	1.04
LS-Altr	1.087	1.06	1.03	1.03	1.03
CPLS-Altr	1.063	1.05	1.02	**1**	**1**
PCCS-$i * k * z$	**1**	**1**	**1**	1.01	1.03

Figure 4.36: PCCS heuristics on random structured graphs.

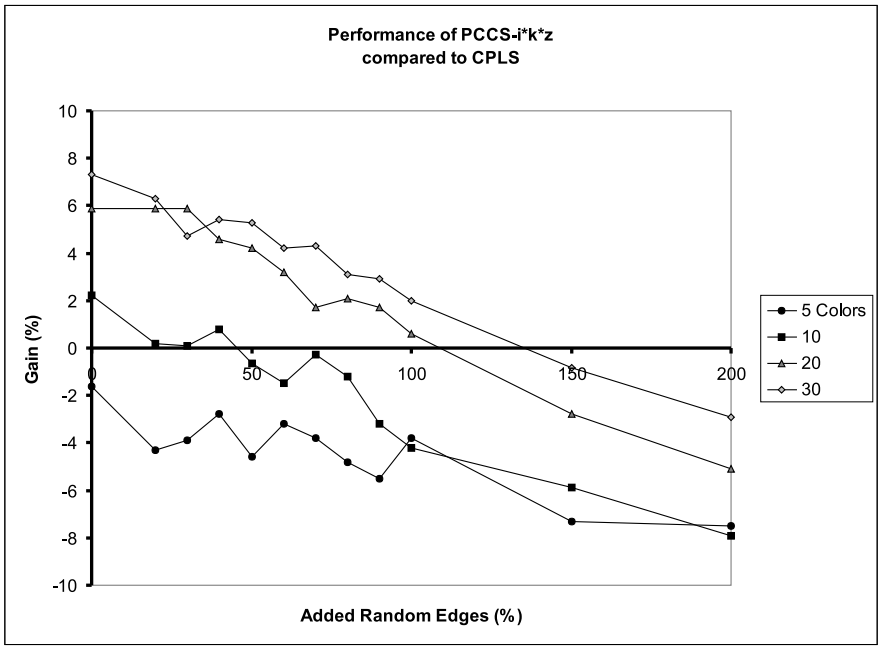

Figure 4.37: OTL Compared to CPLS on random structured graphs.

Evaluation

It is hard to evaluate our heuristics since there is no set of representative instances available. A statistical study of the precedence relation induced by the def-use chains of real $_{\mathrm{d}}$SL programs would have made this study possible. However, to the best of our knowledge, in the literature no such statistical studies have been undertaken for common programming languages. Either research is conducted in the area of statical analysis, in order to extract the precedence relation from a program text (i.e. find the most precise def-use chains), or studies are undertaken to approximate the precedence constrained class sequencing problem with a general precedence relation which is supposed to be available. Little is known about the characteristics of programs conceived by the human mind. We believe however that the class of our instances is very specific, and the particularities of these instances should be exploited to find more efficient heuristics.

With the implementation and the use of the $_{\mathrm{d}}$SL compiler-distributer in the industry, this information could be available in the future, and more precise knowledge could then be collected : both the particularities of the precedence constraints and the distribution of colors on the different instructions could then be studied, resulting in new criteria for our general PCCS-i framework.

Chapter 5

dSL's implementation

In this chapter, we discuss the dSL environment. More specifically, we give implementation details about the different steps taken to transform a dSL source code into an executable code, and show how this code is executed in dSL's execution environment. We conclude this chapter by a critical discussion on the applicability of dSL to complex systems.

5.1 The dSL Environment

5.1.1 Overview

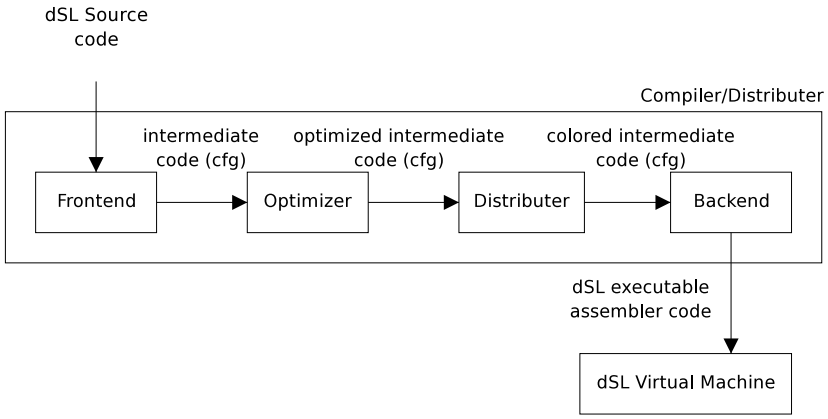

Figure 5.1: Overview of the dSL environment

At first glance, the dSL environment can be divided in two major parts : the dSL compiler-distributer, and the dSL virtual machine. The compiler-distributer takes as input a dSL source code, and transforms it in executable code. This executable code can then be executed on the dSL virtual machine.

We first discuss the compiler-distributer, whose functional blocks are given in figure 5.1. The source code passed to the distributer-compiler is first analyzed by the frontend, which mainly parses the code. Next, a basic jump-analysis and optimization is performed. This is followed by an implementation of the algorithm presented in section 4.3, which assigns all global variables and instructions to the set of sites. Finally, once all code is colored, the backend generates executable code for the dSL virtual machine.

Next, the dSL virtual machine is discussed. This virtual machine is basically a register based CISC architecture interpreting assembler code with some special features such as event handling and inter-site communication primitives.

5.1.2 The Frontend

The frontend is the first part of the dSL compiler-distributer. It parses the source code, and constructs a syntax tree.

While constructing the syntax tree, the frontend checks for syntactical correctness. These correctness checks include variable declaration checking, type checking and sanity of `METHOD` calls and `WHEN`/`WHEN IN` declarations (number of parameters, type of object, the use of declared global variables in `WHEN`, ...). If the code passes these tests, the syntax tree is used to statically transform all `WHEN IN`s into normal `WHEN`s.

Next, the modified program is transformed into a set of control flow graphs (cfg) [ASU86]. There is a control flow graph for each `WHEN`, `METHOD` and `SEQUENCE`. A control flow graph is a graph where nodes are called basic blocks, and edges express the possible ways in which control may flow from one basic block to another. Each basic block in a control flow graph contains a sequential list of instructions. At this point, the possible instructions in the control flow graph are : assignment, call, unconditional jump and conditional jump.

For modularity reasons, the frontend is a separate program. The set of control flow graphs, type and variable declarations are organized in an output file which is fed into the rest of the work flow. A detailed description of this intermediate code representation and the implementation details of the frontend can be found in appendix D.

5.1.3 The Optimizer

The optimizer reads the intermediate code resulting from the frontend. It reads all type and variable declarations, which are put into the compiler-distributer's symbol table. Next, each control flow graph is read and stored

in memory.

Simple jump optimization

First, the optimizer optimizes consecutive jumps and eliminates empty basic blocks. This very simple optimization is not detailed any further.

Code transformation

Next, all instructions are simplified to 3-address code form [ASU86]. Before transformation, all expressions in the assignment instructions and parameters of the call instructions are complex expressions represented as syntax trees. The optimizer transforms these instructions, by introducing temporary variables into simple instructions with at most 3 operands. Figure 5.2 shows how the instruction `x := (y*(x+1))/2;` is transformed in 3-address code.

The use of 3-address code and basic blocks is a standard transformation found in all modern compilers, since it simplifies the static analysis algorithms used in the following chains of the compiler.

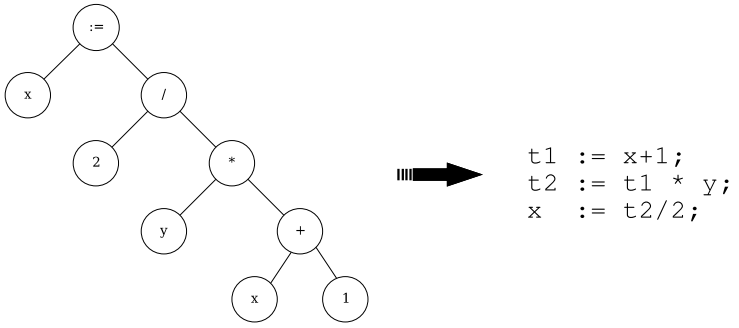

Figure 5.2: Transformation to 3-address code

Specialization of METHODs

With the definition of synchronous flow (cfr. definition 20), many programs are not distributable. Consider for example the following code :

```
CLASS Pump
   current  : INT;
   pressure : INT;
END_CLASS

GLOBAL_VAR
```

```
   pump1, pump2 : Pump;
end_var

METHOD Pump::start()
   self.current := 10;
END_METHOD

WHEN pump1.pressure < 10 THEN pump1<-start(); END_WHEN
WHEN pump2.pressure < 10 THEN pump2<-start(); END_WHEN
```

In this small example, both `WHEN`s make a call to `Pump::start()`, and thus, `pump1.pressure, pump2.pressure, pump1.current` and `pump2.current` are in the same connected component. This is caused by the appearance of `self.current` in `Pump::start()`, which involves variables `pump1.current` and `pump2.current`.

This is an unfortunate situation, caused only by the use of `self` in the `METHOD start()`. Notice that there is no physical constraint forcing all these variables on the same site. What the programmer really wants to express here is the following :

```
// ...

METHOD start_pump1()
   pump1.current := 10;
END_METHOD

METHOD start_pump2()
   pump2.current := 10;
END_METHOD

WHEN pump1.pressure < 10 THEN start_pump1(); END_WHEN
WHEN pump2.pressure < 10 THEN start_pump2(); END_WHEN
```

In this case, `pump1.pressure` and `pump1.current` can be on a different site than `pump2.pressure` and `pump2.current`.

This transformation, called *specialization*, is implemented and automatically applied by the dSL compiler-distributer.

Definition 31 (Specialization) *We define the* specialization *of* `METHOD M` *for object* `O` *as a copy of* `M`*, where all references to* `self` *in* `M` *are replaced by* `O`. ♦

For example, the specialization of `start()` for `pump1` is a `METHOD` containing the instruction `pump1.current := 10`.

The automatic specialization is performed as follows. In general, a call to a `METHOD` can be made in three different ways :

1. The `METHOD` is called using a statically known object.
   E.g. `pump1<-start()` or `pumparray[1]<-start()`.
   In this case, the `METHOD`'s body is specialized for the statically known object, and the `METHOD` call is replaced by a call to the specialized version.

2. The `METHOD` is called using an array.
   E.g. `pumparray[i]<-start()`.
   There are two possible cases.

   - The instruction is in sequential code. In this case, there is a specialized `METHOD` for each possible object used for the call. The instruction is replaced by the following code :
     ```
     IF i == 1 THEN pumparray[1]<-start();
     ELSE IF i == 2 THEN pumparray[2]<-start();
     ...
     ```

   - The instruction is in atomic code. In this case, no specialization is performed, since all objects in the array can be accessed by the instruction, and all calls must end up on the same site anyway.

3. The `METHOD` is called using `self`.
   Since this can only be done inside a `METHOD`, this case is handled when that `METHOD` is specialized. The called `METHOD` is therefore specialized using the statically known object used to replace `self`.

Remark that this procedure was implicitly introduced in the formal semantics of dSL, where we stated that `METHOD`s are inlined in the code. Although specialization does not remove `METHOD`s, it has, with respect to the atomic constraints, the same effect as inlining.

Before the code is passed on to the distributer, the optimizer analyzes the code and performs the necessary specializations. This is a recursive process, since introducing a new specialization for a given `METHOD`, may require more specializations. This is for example the case when inside a `METHOD`, another `METHOD` of the same class is called.

Care is taken that `METHOD`s are not needlessly specialized. In the current implementation, a table keeps track of the different specialized version for each `METHOD`. When the `METHOD` needs to be specialized, the compiler-distributer checks if that particular specialization does not already exists,

reusing existing specializations if possible. This table is currently based upon triples (M, O, M'), where M is the original `METHOD`, O is the object for which M is specialized, and M' is the resulting specialized `METHOD`. If a new specialization is needed, the table is checked and, if possible, M' is reused.

The complexity of the specialization and the transformation of array accesses in sequential code is substantial. Notice that the specialization can be done in linear time w.r.t. the size of the program text and that the number of needed specializations and the number of instructions introduced by the code transformation are both linear in the size of the array. The overall complexity is therefore in $O(|P||A|)$, where $|P|$ is the size of the program, and $|A|$ the number of elements in the array. Nevertheless, if multidimensional arrays are used, the number of specializations may become important, resulting in slow compilation times and large executables.

5.1.4 The Distributer

The distributer is the most elaborate part of the dSL compiler-distributer. It first checks if the program is distributable. Next, it colors all instructions and global variables using the algorithm from section 4.3. Finally, it modifies the control flow graphs by adding synchronization code in the `SEQUENCE`s. This results in a set of colored control flow graphs, which can be used by the backend to produce executable code.

Code details about the distributer can be found in appendix E. Here we discuss, on a higher level, the technical aspects of the last part of the distributer : how colored code is transformed and synchronization code is inserted.

All the technical difficulties come from the use of `SEQUENCE`. As argued before (section 2.2.2), dSL uses thread migration to handle sequential code. Thread migration is used in dSL to obtain data locality. For an instruction to be executable, all its operands (especially variables) must be accessible. The coloring algorithm ensures that each instruction is assigned on the site where its operands are localized. However, consecutive instructions may be located on different sites. When the next instruction needs to be executed on a different site than the current instruction's site, the execution is halted and moved to the other site where execution is continued. Of course, the local context is moved from one site to the other, including local variables and call stack. Classically, the entire thread's local context is moved from one site to the other. In dSL, since the communication layer can be relatively slow, only the necessary part of the local environment is moved, saving bandwidth. The penalty for this compactness is that the distributer must

statically decide what part of the local context is needed. In this section we explain how this information is calculated and how it is translated in the so-called synchronization code.

We first take an intra procedural approach, where the problem of context moving is studied inside a single `SEQUENCE`. Next, we show how this approach can be integrated into an inter-procedural approach, where function calls are taken into account. Dynamic features such as arrays and pointers are discussed in section 7.1.

An intra procedural approach

An illustration of synchronization code inside a simple `SEQUENCE` is given in figure 5.3. In this example, the variables `x, y, w, u, v` are supposed to be local variables, while `Ga, Gb` and `Gc` are global variables localized on respectively site A, site B and site C. Hence the instructions involving `Ga` are localized on site A, instructions involving `Gb` are on site B and instructions with `Gc` are on site C. The transformation of the sequential color graph goes as follows. The initial control flow graph is divided in parts of maximal size such that all instructions in each part are on the same site. Next, on the borders of each such a part, `START` messages are inserted which allow control flow to cross the different sites, in the same manner as if there is only one site. In the example, labels `l1, l2` and `l3` are used to illustrate this. Finally, messages are sent between the parts if a variable is updated in one part and is needed by another part. The extra instructions that are inserted by the distributer at the end of each part are called synchronization code. In the example, three synchronization points are inserted. Remark that the coloring algorithm presented in section 4.3 aims at minimizing the number of synchronization points, weighted by the number of times control flows through such a point.

To formally show how synchronization code is inserted by the distributer, we reuse definition 24 of sequential color graph given in section 4.3.2. We do not need the edges of infinite weight, since we suppose that a correct coloring is already obtained. Since we only look at one `SEQUENCE` at a time, for the time being, intra-procedural edges (introduced by `METHOD` call instructions) are not taken into account either, and only one `SEQUENCE` at a time is considered.

With these restrictions in mind, we can define the following :

Definition 32 (Control Flow Path) *A control flow path is a list of nodes* $n_1, ..., n_k$ *such that* $n_1 \in V \wedge \forall i \in [2, k] : n_i \in V \wedge (n_{i-1}, n_i) \in E$. ♦

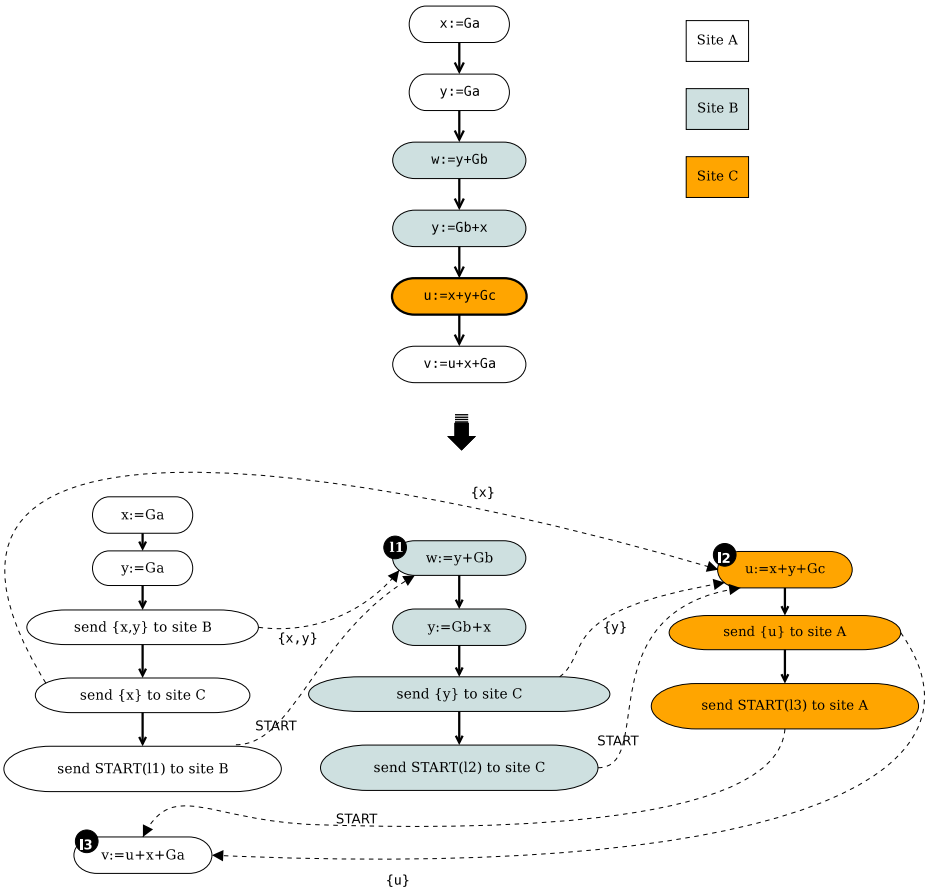

Figure 5.3: Synchronization code inside a `SEQUENCE`

Definition 33 (Live variable) *A variable* `x` *is live in node* n_1 *if there is a control flow path* $n_1, ..., n_k$ *such that* `x` *is used before it is defined in it, i.e.* $\exists i \in [1..k] : n_i = (id_i, m_i, D_i, U_i) \wedge x \in U_i \wedge \nexists j \in [1..i] : n_j = (id_j, m_j, D_j, U_j) \wedge x \in D_j$. ♦

Definition 34 (Definition/Use) *We say that a node* $n' = (id', m', D', U')$ *in the control flow graph uses a definition of* `x` *at node* $n = (id, m, D, U)$ *if there exists a path* $n = n_1, ..., n_k = n'$ *such that* $x \in U' \wedge x \in D$ *and such that* `x` *is live on that path.* ♦

Definition 35 (Def-use chains) *We define function* $DU(n, x)$ *as a function which gives the set of all nodes* n' *such that* n' *uses the definition of* `x` *at* n. ♦

Def-use chains are easily obtained by using classical static code analysis, and are widely used in optimizing compilers [ASU86]. The algorithms calculating $DU(n, x)$ are defined using backwards passes on the basic blocks in the control flow graph. They allow the compiler to perform (amongst others) dead code elimination, register allocation and constant propagation. Here, we use def-use chains to know on which site a live variable may be used in the future.

We define the set of nodes where synchronization code has to be inserted (synchronization points) as follows.

Definition 36 (Monochrome set) *A monochrome set of a node* n *is the maximal set of nodes reachable from* n *in the sequential color graph, having the same color, i.e.* $MC(n) = \{n_k \in V | n = n_1, ..., n_k$ *is a control flow path and* $\forall i \in [1..k] : c(n) = c(n_i)\}$. ♦

Definition 37 (Exit boundaries) *The exit boundaries of a monochrome set* $MC_{\rightarrow}(n)$ *is the set of nodes in* $MC(n)$ *having at least one successor with a different color, i.e.* $MC_{\rightarrow}(n) = \{n' \in MC(n) | \exists (n', n'') \in E \wedge c(n') \neq c(n'')\}$. ♦

To explain how the message constructing algorithm works, consider a node n in the sequential color graph, that defines a variable `x` (for example `x := y + z;`). For this node, we can use $DU(n, x)$ and obtain the set of nodes which use this definition of `x`. For each such a node n', on a different site than n, we have to send the new value for `x`. The total number of messages to send is the number of colors of the nodes in $DU(x, n)$ that have a different color than $c(n)$.

Two strategies are possible : we can either send the value of `x` at once, or we can postpone the message until we actually leave the current site. We choose the more efficient second solution. The benefit of this approach is that bandwidth on the communications network is lower, and processing power for the composition and emission of messages is spared. Indeed, consider for example a loop that constantly changes `x`, while remaining on the same site. It is easy to see that sending a message each time `x` changes will introduce a severe performance penalty.

Clearly, the transmission has to be done in one of the boundary nodes of the monochrome-set started at n. Indeed, the nodes in $MC_{\rightarrow}(n)$ are exactly those nodes that leave the current site, but not all of those nodes may lead to a use of `x`. In fact, because we postpone the transmission, we can end up in a situation where part of the calculated def-use chains do not hold anymore. This is the case for $\{n_1, n_2\} \subseteq MC_{\rightarrow}(n)$ where x is not live in either n_1 or n_2. An example of this case is presented in the right branch of figure 5.4, where x is a variable not live in n_2. Note that for each given destination site (the color of a node in $DU(n, x)$ is different from $c(n)$) at least one boundary node exists where the transmission must be made (if not, the set $DU(n, x)$ is erroneous). To avoid such undesirable cases, we transform the sequential color graph and insert dummy nodes (where $D = U = \emptyset$) in order to have for each node in $MC_{\rightarrow}(n)$ only one successor. For each node n' which has multiple successors n_i (all with different colors than $c(n)$), we insert a dummy node d_i, remove the edge (n', n_i) and insert edges (n', d_i) and (d_i, n_i).

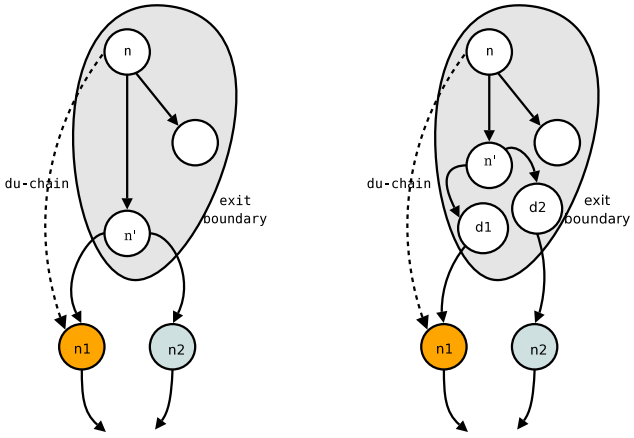

Figure 5.4: Postponing transmissions

Finally, observe that if there is a node $n' \in DU(n, x)$ with $c(n) \neq c(n')$,

then for all paths $n = n_1, ..., n_j, ..., n' = n_k$ where `x` is live, we can find j such that $c(n_1) = c(n_2) = ... = c(n_j) \neq c(n_{j+1})$. Remark that n_j is in $MC_{\rightarrow}(n)$ and that $n' \in DU(n_j, x)$. In other words, all information to construct correct synchronization code is contained in the def-use chains at the boundary nodes. The algorithm inserting synchronization code can therefore be limited to the inspection of all def-use chains at the exit boundary nodes.

The synchronization code emitting algorithm is therefore extremely simple :

for each n `such that` $\exists(n, n') \in E \wedge c(n) \neq c(n')$ **do**
  for each $n', x \in DU(n, x)$ `such that` $c(n) \neq c(n')$ **do**
    `EMIT send x to` $c(n')$
  end for each
  `EMIT start label(`n'`) to` $c(n')$
end for each

Restrictions on communications

Two remarks should be made concerning the previous algorithm. First it is clear that all communications must be made over reliable channels. If a message is lost, some updates will not be received and old values for local variables may be used instead of the correct values.

Second, the order in which the messages are sent must be the same as the order in which messages are received. Indeed, consider the example of figure 5.5, where `x, z` are local variables and `Ga, Gb, Gc` and `Gd` are global variable localized respectively on site A, B, C and D. The use of `x` in node n' will depend on the execution. Therefore, the message from A to D should be received before the message from B to D. If this is not the case, D will use a stale version of `x`.

This ordering can be obtained easily if acknowledgments are awaited for each message before sending the next one.

An alternative is to use message clocks, inspired by vector clocks [Mat89] which are used in distributed systems to obtain a logical clock. A message clock $\overline{c}$ is a vector of n counters where n is the number of processes in the distributed system. Each process i maintains one message clock $\overline{c}_i = (c_{i,1}, \ldots, c_{i,n})$. The algorithm goes as follows :

- Initially all clocks are set to 0
- Each time a process i sends a message to j, before sending the message,

it increments its counter $c_{i,j}$ by one and appends the message clock $\overline{c}_i$ to the message.

- Each time a process i receives a message from j with message clock $\overline{c}_j$, it checks that $c_{i,i} = c_{j,i} - 1$. If this is condition does not hold, the message is suspended for later treatment. If the condition holds, the process accepts the message and updates $\overline{c}_i := \overline{c}_j$.

The entry $c_{i,i}$ maintains the number of messages received by process i, while $c_{i,j\neq i}$ keeps track of the number of messages sent to j by process i. Note that if a process can send messages to itself (which is the case in dSL), a separate counter in each process should be used instead of $c_{i,i}$ to keep track of the number of received messages.

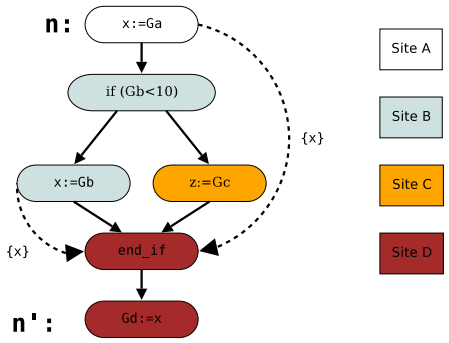

Figure 5.5: Message ordering must be respected

Code details about the distributer can be found in appendix E.

Interprocedural approach

Now that we know how to efficiently move execution from one site to another inside a `SEQUENCE`, we can discuss how `METHOD`s are handled.

Two important restrictions are made upon the coloring of sequential instructions that simplify the execution of `SEQUENCE`s with `METHOD` calls.

The first restriction concerns the first and last instruction of each `METHOD` with sequential code, which must be of the same color. To ensure that this is the case, a dummy instruction with the color of the first instruction is added at the end of each `METHOD`, for `METHOD`s where the coloring algorithm did not come up with such a coloring.

The second restriction makes `METHOD` calls always happen on the same site as the first instruction of the called `METHOD`, and must be followed by an instruction on that site. Note that, by the previous restriction, this is

also the same site as the last instruction of that `METHOD`. This hypothesis can easily be fulfilled by inserting a dummy instruction after the call, and instructions with the desired color before each `METHOD` call, not respecting this restriction. If no parameters are passed, these instructions amount to a single dummy instruction, if parameters are passed, each parameter is stored into a local variable created for that purpose. In this way, the `METHOD` call, with the local variables as parameters, can entirely happen on the site on which the first and last instruction of the called `METHOD` are localized.

We already argued that memory allocation is done statically in dSL, which results in the restriction that only one instance of a given `SEQUENCE` can be active at a certain time. A statical, interprocedural, analysis of the code assembles the set of sites on which a given `SEQUENCE` may execute. For each `SEQUENCE` and each site on which it executes, memory is allocated by the dSL virtual machine to contain the execution stack of that `SEQUENCE`.

When a `SEQUENCE` is started, all intervening sites are informed by messages over the network. These messages contain the unique identifier for that `SEQUENCE` and are sent from the site which contains the first instruction of that `SEQUENCE`. Upon reception of such a message, the receiving site uses the previously allocated memory to initialize the stack pointer and execution stack for that `SEQUENCE`. From that point on, all sites can receive local variables by the mechanism described before.

When a call instruction is encountered on a certain site, all sites on which the `SEQUENCE` may execute are informed to create a stack frame for the called `METHOD`. This is done by sending messages to all concerned sites, containing the unique identifier for the `SEQUENCE` and the `METHOD`. Next, the `METHOD` call is executed. This means that the address of the next instruction is pushed upon the stack of the current site, and parameters, which are local variables inside the called `METHOD`, are inserted in the newly created stack frame. Finally, the execution pointer is updated to the first instruction of the called `METHOD`. Remark that all sites are now ready to receive local variables if execution passes through them.

When the end of a `METHOD` is reached, the return address is fetched from the current stack. Remark that this is only possible because of our two initial hypothesis. Next, all concerned sites are asked to remove the last stack frame. Finally, the stack frame on the current site is removed as well, and the instruction pointer is updated according to the fetched return address, which is an instruction on the same site.

A small example of this mechanism is given in figure 5.6. In this example, global variables `Ga, Gb` and `Gc` are localized on site A, B and C respectively, and a `SEQUENCE S` calls a `METHOD M`.

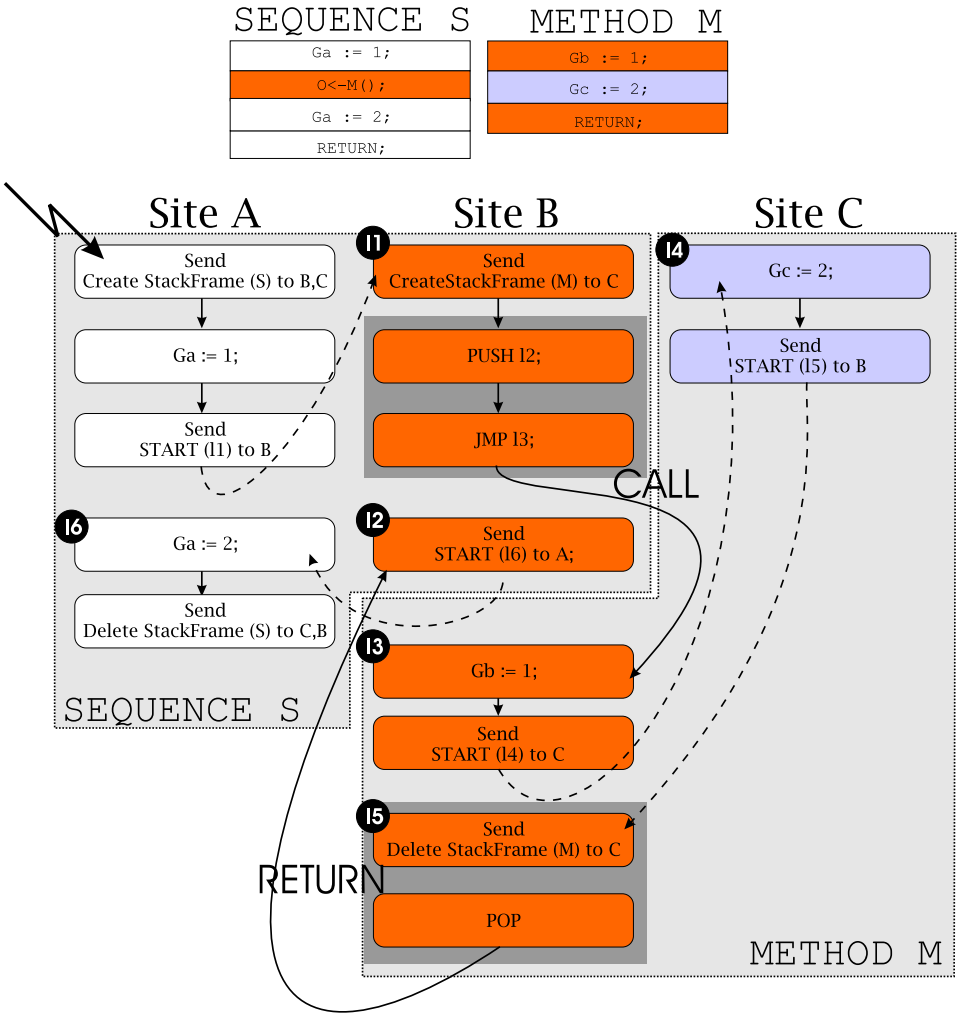

Figure 5.6: Illustration of the interprocedural approach.

5.1.5 The Backend

The backend is the last part of the dSL compiler-distributer. At this point, all instructions are in 3-address form, colored, and synchronization code is generated.

The task of the backend is to produce executable code. Three different versions of the backend have been implemented, one for the dSL virtual machine platform, which is used in the industry, one for the production of PROMELA code, which can be used for the automated verification of dSL, and one for the production of LEGO MINDSTORM code, which is used for educational purposes. The translation to PROMELA is discussed in the next chapter. The LEGO MINDSTORM backend and its execution environment is presented in [Dev04].

The dSL Virtual Machine backend is much like any classical compiler backend. The backend produces a set of files, one for each site in the system. Each file contains type definitions, global variable declarations, and executable code. It manages register allocation by graph coloring [Bri92] and liveness analysis. Stack frame organization is based upon the ideas presented in [ASU86]. The production of executable code from 3-address instructions in control flow graphs is straightforward, and also presented in [ASU86]. Details about the file format produced by the backend and code details can be found in appendix F.

5.1.6 The dSL Virtual Machine

The dSL virtual machine has been designed as a compromise between performance, debugging possibilities and ease of communication between different Virtual Machines. The high performance requirements result in a register based machine, as opposed to a stack machine. The use of CISC 3-operand instructions makes it possible to do a step-by-step debugging. The communication primitives all use unique global identifiers for `METHOD`s, `SEQUENCE`s and global variables, which are translated into addresses in local memory for those objects which are accessible on a given virtual machine.

The dSL Virtual Machine is developed for execution in the environments AIX/PowerPC, PC using Linux and Windows. It implements a 32-bit CISC architecture and uses registers.

The Virtual Machine's Context

The dSL Virtual Machine's main target platform is an industrial controller, which is never halted. For this reason, each Virtual Machine contains two

contexts : the running context, and the shadow context. A virtual machine which is executing its running context is able to load a new program in its shadow context. When the shadow context is ready for execution, it can become the new running context. By this mechanism, a new version of a program can be prepared offline, and almost instantly activated.

The context contains all objects which are handled by a virtual machine : type definitions, variable declarations, constants, code segments, events, sequences and I/O definitions.

The type definitions which are currently in use by the dSL virtual machine include all basic types `BOOL, DINT, SINT, REAL, DATE`, ... and more complex types such as `ARRAY` and `STRUCTURE`.

A global variable in dSL is identified with a unique distributed identifier (a 32-bit integer), which is attributed by the dSL compiler-distributer. For each global variable, the Virtual Machine maintains a list of events (cfr. `WHEN`s) for which the variable appears in the condition, and a list of sites to which the Virtual machine has to send updates when the value of the variable changes (cfr. the $\sim$ operator). These lists are passed to the Virtual machine by the compiler-distributer.

Constant definitions are treated in the same way as global variables, with the difference that they cannot be affected to.

A code segment is basically a list of dSL assembler instructions. Each code segment is identified by a unique distributed identifier, which allows one virtual machine to start code on another virtual machine. There is a segment for each `WHEN` condition and body, for each `SEQUENCE` and each `METHOD`. The instructions inside a segment are coded as 4 words of 32 bits of the form `<Opcode, Operand 1, Operand 2, Operand 3>`.

Events are triggered by the change in some global variable's value. There is an event for each `WHEN` in the program source. Each event has two segments of code to which it is linked : a segment which evaluates the `WHEN`s condition, and a segment containing its body. In addition to those segments, the Virtual Machine allocates memory to store the previous value of its condition. This allows to execute the body of an event only when a rising edge occurs. When a new value is assigned to a global variable, the list of events is used to check for a rising edge in the event's condition. When such a rising edge occurs, the execution is interrupted, and the event's body is executed. An event stack is used to handle recursive triggerings.

Tasks[1] are the executable processes in a Virtual Machine. A task is also identified by a unique distributed identifier. Each task contains an execution stack and a segment, containing its code. There is at least one task on each virtual machine, which is called the 0-task. It is used to handle events caused by changes in external and ∼ variables and `LAUNCH`ed code.

I/O variables represent the external global variables in the dSL program. Input variables can only be read. They contain the `UNKNOWN` value when the hardware to which they are attached is malfunctioning. Output variables can only be written. When a `UNKNOWN` value is written to them, a predefined neutral value for that output is copied to the hardware. I/O variables can be binary in case of digital circuitry, or integer in case of analog hardware.

General Behavior

The dSL Virtual Machine executes an infinite loop, which performs two tasks.

The first task for the virtual machine is to listen for incoming configuration messages. These messages are used for debugging purposes and allow the initialization and activation of the shadow context.

The second Virtual Machine's task is to execute one Input-Process-Output cycle. Each such cycle performs the following four steps :

1. Copy all hardware inputs to the associated external variables (no events are triggered).

2. Trigger events caused by the new state of the external variables.

3. Inspect and handle pending incoming messages. Incoming messages may be of tree kinds: ∼ messages, `LAUNCH` messages, or `SEQUENCE` messages.

4. Copy all external output variables to the associated hardware devices.

A ∼ message contains the updated value of a distant global variable for which a tilded version exists on the current site. It is sent by that distant site because the value of the distant variable changed. The treatment of such a message consists in assigning the new value to the tilded local version, and triggering events if necessary.

[1] In the Virtual Machine documentation, the word *sequence* is used to refer to the processes inside the Virtual Machine. To avoid confusion with the language construct `SEQUENCE`, we refer to them as tasks.

A `LAUNCH` message asks for the execution of some code segment which is localized on this site. The treatment of such a message ends when the activated code returns from execution.

A `SEQUENCE` message is related to synchronization code, or stack-frame creation/deletion. Treatment for this messages is discussed in section 5.1.4. Remark however, that when the message contains a `START` command, the corresponding segment and label is activated, and treatment ends only when the `SEQUENCE` ends its execution, or switches to another site.

Note that all steps of the Input-Process-Output cycle, except the treatment of `SEQUENCE` messages, happens inside the previously introduced 0-task.

Note also that no multitasking is needed inside the dSL virtual machine. This design choice increases performance stability, simplifies the implementation and eases debugging.

5.2 Relation to the formal semantics

It is clear from the formal semantics that the implementation of dSL cannot, and should not, include all behaviors given in the formal semantics of dSL. For example, the implementation has limited memory, while the formal semantics describes an infinite state system. This is the case for example when recursive `WHEN` triggerings occur. Another difference is that the formal semantics describes the system as a composition of fully asynchronous processes, whereas the implementation respects some synchronization due to timing constraints : the cycle time of each site in the implementation, for example, will be in a given bounded range, which is not modeled in the formal semantics.

The point is that the formal semantics describes a set of behaviors that must be a superset of the set of possible behaviors of the real implementation. In other words, anything the implementation can do must be captured in the formal semantics.

It is hard to formally show that this is the case for dSL (or any complex programming language), since a formal proof showing that the possible behaviors of the implementation are contained in the behaviors given in the semantics, would require a formal description of the implementation, which raises the same question. For this reason, we do not give a formal proof here. We will give solid arguments for each rule in the semantics showing that the implementation respects the semantics.

- **[Interleaving]**

The interleaving rule states that whenever a process can make a move, the global state of the system can make the same move.
This is obviously the case for the implementation since no blocking synchronization whatsoever is used in the implementation.

- **[Broadcast]**
  The broadcast rule instantly inserts messages in the FIFO channels of all processes.
  This is not the case in the implementation, since an asynchronous network is used, and messages are sent one after the other. However, in the semantics, these messages can be picked in any order, and may take an arbitrary amount of time before being inspected by the processes. This correctly models the asynchronous network used in the implementation

- **[Cycle start]**
  The cycle start rule schedules a new cycle, consisting of

  1. Input
  2. Treatment
  3. Message Treatment
  4. Output

  Moreover, the different (sub)channels of the process starting a new cycle are frozen by the insertion of delimiters $\diamond_i$.
  In the implementation, an infinite loop schedules the same actions. The insertion of the $\diamond_i$ markers are automatically handled by the execution environment, which does not accept new messages in the queue[2].

- **[Input,Output]**
  The input and output rules express the interface with the environment. These rules are handled by the underlying hardware of the execution environment, and behaves exactly the same.

- **[Message treatment]**
  The message treatment rule describes how messages in the (frozen) FIFO channel are removed, from any sender, respecting the order in which they where sent. Each message updates the value of a tilded

[2] in fact, it is the underlying operating system that buffers incoming messages during the cycle

variable, and WHENs are triggered if necessary.
The implementation takes messages one by one, from any site, and for a given site, in the order in which they where sent by that site. The behavior is the same (the variable's value is updated and WHENs are triggered)

- **[End of message treatment]**
  The end of message treatment rule can be fired when no more messages are in the (frozen) FIFO queue.
  The same happens in the implementation.

- **[Assignment], [If]**
  It should be clear that these rules are respected by the implementation

- **[Sequence Activate(′)]**
  The sequence activate (activate′) rules are triggered when a SEQUENCE is LAUNCHed by a process. σ_s is updated to contain the instructions to execute and ξ is set to any site capable of executing the first instruction and μ is updated for all local variables inside the SEQUENCE.
  In the implementation, SEQUENCEs are not global objects. Instead they are distributed amongst the different sites. Each instruction is assigned to a certain site, using the the algorithm described before. The LAUNCH instruction causes an activation message to be sent to the site containing the first instruction of the SEQUENCE. Since this site is clearly in $Exec_D(\sigma_s)$, this is consistent with the semantics. Note that the destination site does not receive this message immediately, which is not the case in the semantics where the change of ξ is instantly visible to all processes. However, the semantics for the execution of SEQUENCEs states that the process can wait an arbitrary amount of time before executing the SEQUENCE. This models the fact that the activation messages takes time to travel to the destination site.
  The update of μ is respected by the implementation, as described in the intra-procedural approach of section 5.1.4.

- **[Sequence Assign]**
  The sequence assign rule can be executed when a process is in its message treatment phase. It performs the assignment from a SEQUENCE, sends updated values and triggers WHENs if necessary. μ is updated if the left hand side is a local SEQUENCE variable, and the SEQUENCE is migrated to any site capable of executing the next instruction.
  The implementation handles the assignment in the exact same way (sending messages and WHEN triggering). The difference resides in the

fact that μ is not instantly known to all sites. Instead, synchronization code sends messages to update the value of local variables from the `SEQUENCE`. Another difference is that the `SEQUENCE` cannot instantly migrate in the implementation since an activation message is sent instead. The reasoning is the same as the one held above : in the semantics, processes can postpone the execution of the `SEQUENCE` which models the time needed for the activation message to reach the destination site. The fact that μ is not updated instantly has no effect, since the messages containing the new values of the local `SEQUENCE` variables will be treated before the activation message, as argued in section 5.1.4.
Again, all instructions are localized at compile time, the `SEQUENCE` can only migrate to a single destination site (i.e. the one on which the next instruction is localized). But since this site must necessarily be part of $Exec_D(\sigma'_i)$, this behavior is included in the semantics.
Remark that if the `SEQUENCE` remains on the same site, no migration is done, and the site immediately executes the next instruction. This is also possible in the semantics when $\xi' = \xi$. The next instruction can then immediately be treated by firing the *Sequence If, Sequence While* or *Sequence Assign* rule.

- **[Sequence `IF`, Sequence `WHILE`]**
  The reasoning is the same as the one held above.

5.3 The real $_d$SL environment

The prototype $_d$SL compiler-distributer has been taken over by Macq Electronique, and is integrated in their tool worksuite called OBViews. This worksuite consists (amongst others) of a database, a history server, a synoptical viewer and a programming console. These tools run on a server, which is connected to the network of `PLC`s, as depicted in figure 5.7.

The database

The database is the main source of information for the $_d$SL compiler-distributer. It contains all type definitions, all global variables (with their types), kind (internal, external input or external output) and their localization if applicable. When a $_d$SL program needs to be compiled, this information is extracted from the database and is passed to the $_d$SL compiler-distributer. A graphical user interface (cfr. figure 5.8) enables access to the database.

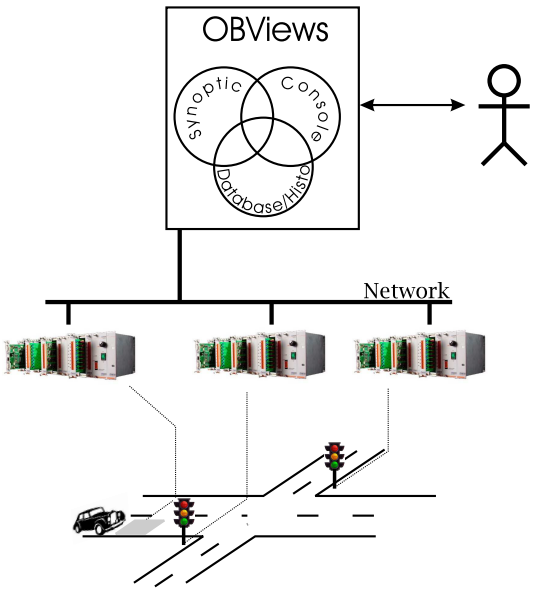

Figure 5.7: dSL and OBViews

The console

The programming console is used as a management tool for dSL programs. The console gives access to the database, the dSL compiler-distributer, the dSL virtual machines and the PLC hardware. A screenshot of the console is given in figure 5.9.

A graphical user interface allows one to remotely administrate the different PLCs connected on the network. This part of the console gives access to the configuration of all input/output cards on a given PLC. The tool can also be used to make a quick diagnostics of the status of a certain PLC.

The console also allows one to browse the database, and to compile a dSL program. If the compilation terminates successfully, a set of executables are ready to be sent over to the different PLCs; otherwise, the user interface displays the compilation errors to the user.

Additionally, the console allows to load the different executables into the set of PLCs. The set of executables is sent over the network to the dSL virtual machines, which load them into their shadow context. Once all the codes are in place, the console can be used to make the dSL virtual machines switch from their running context to their shadow context.

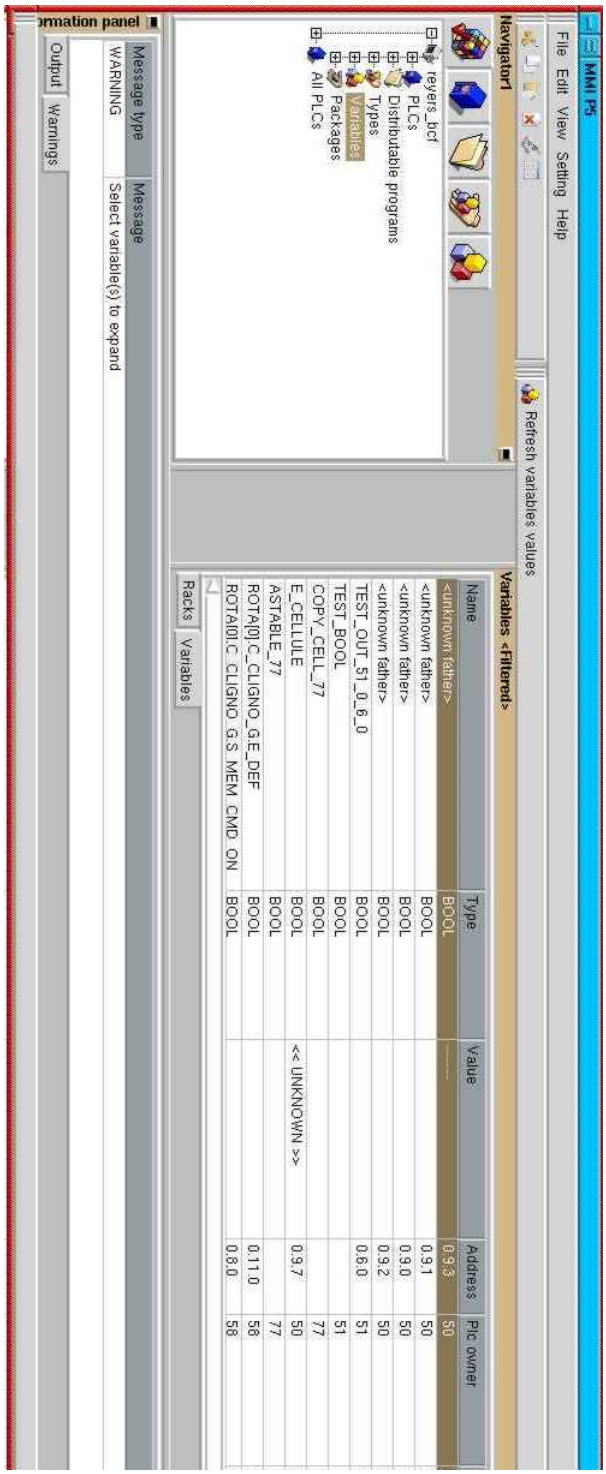

Figure 5.8: Interface to the Database

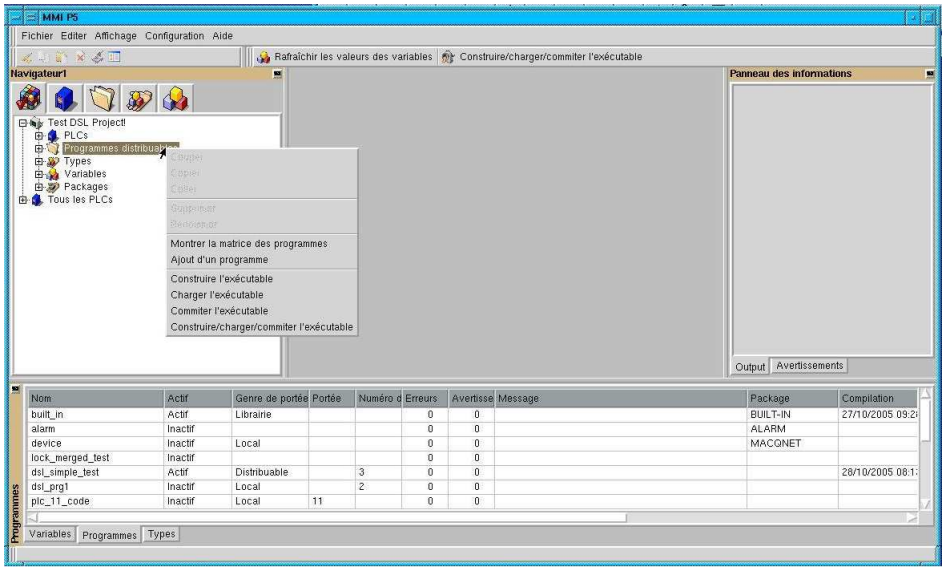

Figure 5.9: The Programming Console

The History Server

The history server is responsible for maintaining the relevant changes in the dSL global variables.

The interactions between the history server and the dSL virtual machines are implemented using messages which travel over the network. These communications are completely asynchronous, and for each global variable can be configured to use one of the following modes :

- Not relevant : The value of the variable is kept in the memory of the dSL virtual machine, and is never communicated.

- By value : The current value of the variable is sent over to the history server when the dSL virtual machine is asked for it. The history server only periodically asks for the value of the variable in order to limit communication bandwidth. Remark that when this mode is used, not all changes are guaranteed to be *observed* by the viewer.

- By perturbation : When the variable's value changes, the dSL virtual machine appends the triple (x, v, t) to a buffer in memory, where x is the variable, v its value and t a time stamp recording the time at which x was changed. When the buffer is full, or when a timeout expires, the buffer is sent over to the history server.

The Synoptic Viewer

The synoptic viewer allows the end user to supervise and interact with the controlled system. It displays a synoptic, which is an animated, interactive graphical representation of the real system. An editor allows to create these synoptics, and allows - through the database - to link the graphical animations with the state of the global variables in the dSL program. Additionally, the synoptic can contain interactive graphical objects which allow to update the value of some global variables in the dSL program. A synoptic can therefore be seen as a part of the environment which, in contrast with the PLCs controlling industrial equipments, interacts with the user.

An image of a synoptic which is used to control a complex crossroad using dSL, is given in figure 5.10.

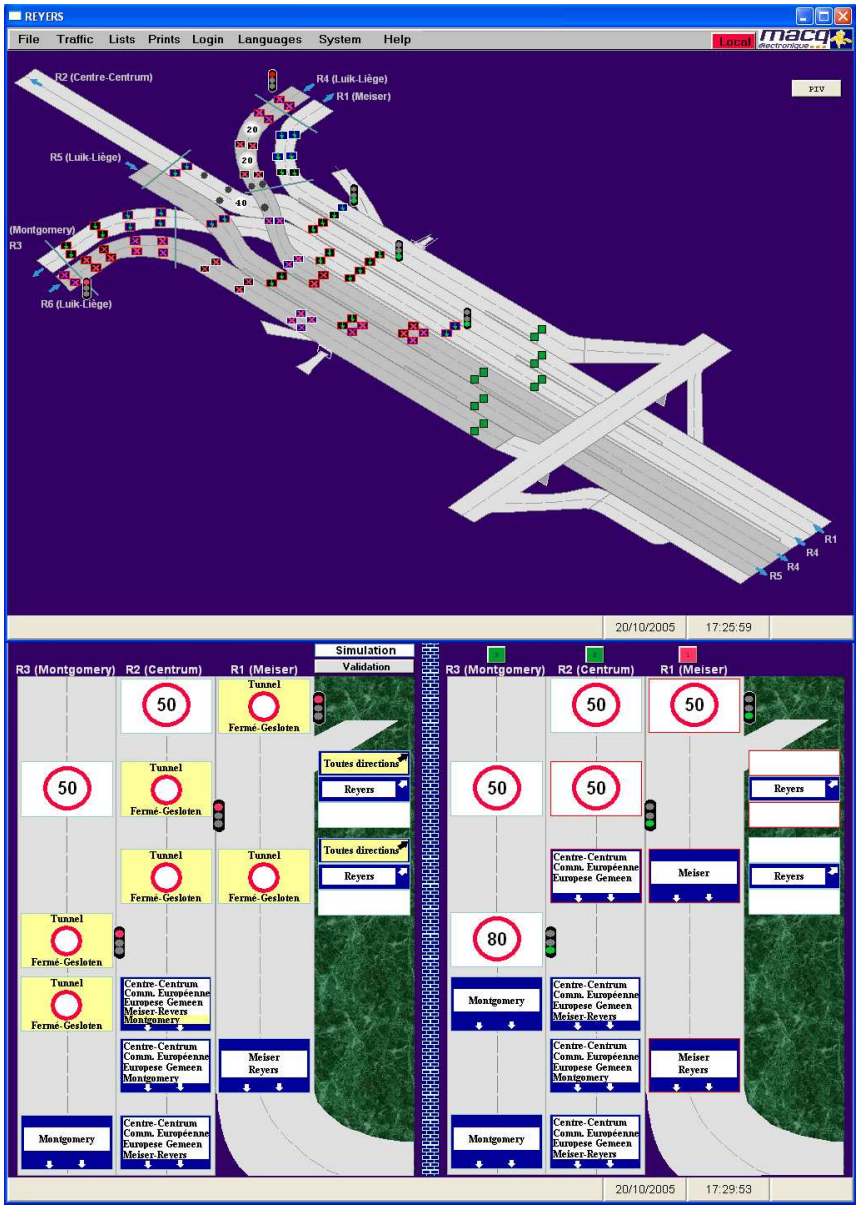

Figure 5.10: A synoptic for a crossroad.

Chapter 6

Verification of dSL

6.1 The *Spin* Model Checker

A model checker [EC99] is a software tool used to formally verify the correctness of a certain specification. A specification is usually an abstracted presentation of the real system, describing the behavior of the *suspect* part of the real system. The abstract model ($\mathcal{M}$) is then used by the model checker to verify whether a certain property (Φ) holds or does not hold in the model.

$$\mathcal{M} \models \Phi$$

Properties are often specified in temporal logical formulae, such as LTL or CTL [Eme90], and can be divided in two sets. On the one hand, one can ask questions on the reachability of some state in the model. These properties are called safety properties. A typical example of a safety property in LTL is `[] ! bad`, expressing that a certain bad condition never happens. On the other hand, questions about the responsiveness of the system can be asked, in the form of liveness properties. A typical example in LTL is `[] (req -> <> ack)`, stating that whenever a request is made, an acknowledge is followed at some point in the future. For an in depth study of these two kinds of properties, we refer the reader to [NC00].

The model checker terminates either with a positive answer, which means that the model satisfies the property, or with a negative answer together with a counter-example showing how the model violates the property.

To establish the (in)correctness of a model, the model checker performs an exhaustive exploration of the state space of the model. There are basically two different types of model checkers, based on the way they represent states. Explicit state model checkers represent each state explicitly, while symbolic

model checkers use a symbolic representation of sets of states[McM93]. Explicit state model checkers are therefore limited to finite state models, while symbolic model checkers can be used to establish the correctness of infinite state systems such as real-time systems [HHW97, BLL+96] or systems with an unbounded number of processes [Beg03].

In this thesis, we use *Spin* [Hol03], an explicit state model checker, which uses LTL to describe properties, and PROMELA as a specification language.

The underlying mathematical concepts used in the *Spin* model-checker are based on communicating finite state machines [BZ83]. These finite state machines are specified by the user in a compact description using the specification language PROMELA. When the model is checked for correctness with respect to a certain property Φ, it is first translated into a Büchi automata ([Buc60]) $\mathcal{S}$. Next, the resulting automata $\mathcal{A}$ of the product of all concurrent finite state machines is calculated. Each run in $\mathcal{A}$ can be seen as a possible execution of the system. All possible executions of the system are therefore captured in the language produced by $\mathcal{A}$. If these behaviors are a subset of all behaviors described by the property, we can conclude that the system is correct with respect to that property. In short, the system is correct if $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{S})$. Let $\overline{\mathcal{L}(\mathcal{S})}$ be the inverse language; we can now reduce the verification to checking the following equation

$$\mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(\mathcal{S})} = \emptyset$$

In order to verify a system with this approach, one must be able to (1) complement $\mathcal{S}$, that is, construct an automaton $\overline{\mathcal{S}}$ that recognizes $\overline{\mathcal{L}(\mathcal{S})}$ and (2) construct the automaton that accepts the intersection of the languages $\mathcal{L}(\mathcal{A})$ and $\overline{\mathcal{L}(\mathcal{S})}$.

In *Spin*, the first operation is done by the user, who has to specify the *bad* behaviors as an LTL formula. The second operation requires to perform the product of the property automaton and the system. However, if the complete automaton $\mathcal{A}$ has to be calculated prior to the verification, it may result in an automaton that is exponentially bigger than the system's description. To avoid this, the product of all communicating processes and the property is calculated on-the-fly. Therefore, an error may be found before the complete state space is constructed. Another advantage is that not necessarily all states in $\mathcal{A}$ have to be constructed to prove correctness of the system [CVWY92].

In addition to its on-the-fly model checking algorithm, the *Spin* model checker uses a partial order reduction algorithm during verification. Partial order reduction methods [Val91, Pel94, God91] are based upon the observa-

tion that in concurrent systems, many runs are equivalent with respect to some property, i.e. two or more runs cannot be distinguished by the property. This is because the property is often insensible to the order in which the different events in the run happen. If this is the case, and the events are independent, i.e. switching the order in which they take place leads to the same state, then only one order has to be checked. The results of applying this method when verifying the system may cause an exponential reduction of the explored state space during verification.

In conclusion, we can state that *Spin* is an explicit on-the-fly finite state model checker. It uses communicating finite state machines specified in PROMELA as specification language, and LTL as a language to specify properties. During verification, partial order reduction is used to tackle the state space explosion problem.

For a short overview of PROMELA, we refer the reader to appendix G. In what follows we show how PROMELA can be used to capture the behavior of dSL programs (section 6.2). We then give some results for the examples given in chapter 3.

6.2 Translation of dSL to PROMELA

The semantics of dSL usually allows an almost immediate translation of a given dSL program into PROMELA. Remark, however, that this translation depends on the distribution used to run this program, meaning that the PROMELA specification will change every time another distribution is used.

For a given distribution, the number of sites, and for each site, the global variables and instructions on that site are known. We translate each site as a `proctype` in PROMELA. Each such a process executes the Input-Process-Output phases presented earlier. The Input and Output phases have a direct link to the environment, while the process phase uses message channels as specified by the semantics of dSL.

In this section, we first give the restrictions imposed by the use of PROMELA. Next, we describe how to efficiently model the environment. Finally, we detail the translation of the code for the different dSL sites into PROMELA. The dSL compiler-distributer's PROMELA backend performs this translation automatically for programs without `SEQUENCE`s, except for the environment which must still be specified manually. To model the environment, the programmer (1) must be able to specify non-deterministic behavior (2) have the possibility to inverse the function of inputs and outputs (inputs for the system would become outputs for environment and vice versa) (3) must have a possibility to specify instantaneous reactions, even for

distributed inputs and outputs. Those 3 elements are not available in $_{\mathrm{d}}$SL, and the designer must therefore encode the environment's behavior directly into the specification. A preliminary study on how $_{\mathrm{d}}$SL could be extended to allow the specification of the environment can be found in [God05].

6.2.1 Limitations

Translating full $_{\mathrm{d}}$SL into PROMELA is difficult for two reasons. First, recursive functions are difficult to translate into PROMELA. However, recursion is generally not a desired feature for industrial controllers. The second difficulty arises from the restrictive finite state space representation used by the *Spin* model checker which imposes a bound on the sizes of the communication channels. Static analysis techniques [LMW04] or testing [Jér91, JJ93] can be used to detect problematic systems, where no such bound exists.

The constructs omitted in the semantics presented in section 3.4 (`LAUNCH`, `WAIT`) can easily be translated, either by replacing them using the constructs that are included in the semantics, since they do not contribute to the expressiveness of the language, or by direct translation into PROMELA.

Modeling the environment

We have several solutions to model the environment of a $_{\mathrm{d}}$SL program in PROMELA. We could consider the environment as an individual process that reacts on outputs and continually changes the inputs of the $_{\mathrm{d}}$SL program. But this approach is quite inefficient. Indeed, consider the behavior of a process P (site) in the system, and more particularly its infinite *Input-Process-Output* loop. Since inputs are sampled at the beginning of such a cycle, and outputs are written at the end, the changes of the environment during the cycle have no effect whatsoever on the process phase of P. Thus, to avoid unnecessary interleaving between the environment and the different $_{\mathrm{d}}$SL processes, the part of the environment connected to this process should be *frozen* as long as process P is in its process phase. To achieve this, we integrate this part of the environment into the specification of process P (by means of *inlining*). Doing so allows the environment to change state only when the process reaches its input phase. The communications between the process and its part of the environment is done through shared variables (representing the I/O variables).

Modeling the processes

The processes described in the semantics can be coded almost as-is into PROMELA processes using a straightforward syntactical transformation. Indeed, each $_{d}$SL process can be translated into a PROMELA process that consists of an infinite loop performing the following steps : *input-process-output*. There is one important difference between the formal semantics and the PROMELA model, concerning the FIFO channels. In order to keep the semantic rules as concise as possible, we introduced only one FIFO channel for each process, and tagged each message with its sender. In the PROMELA model, we introduce a FIFO channel, `chan_r_s`, for each receiving process `r` and sending process `s`.

The input and output phases are interactions with the environment as described above. The process phase does the following steps :

- Input treatment : the triggering conditions of all `WHEN`s owned by the process are considered consecutively successively in order to react to the input changes.
- Message treatment : reads the messages sent by other processes. This message treatment is translated into a loop reading a non-deterministic number of messages from the input channel for this process. Every time a message is read, `WHEN`s depending on the updated variable are checked for execution.

For this translation to work, we must have a special treatment for all assignments in the $_{d}$SL program. Hence, an assignment `x:=e;`, executed by process s, is translated into PROMELA as follows :

- `x:=e;`: the assignment is also performed in PROMELA.
- `chan_1_s!x,e ..., chan_n_s!x,e`: broadcasts the new value to all sites (1..n) that use `~x`.
- consider all `WHEN`s conditioned on `x`.

Figure 6.1 shows the general skeleton in PROMELA of the behavior of a site `X` in the system.

According to the formal semantics, communications between the different processes are modeled using PROMELA's `chan` and are kept reliable but not instantaneous. They have a maximal size `MAX_CHANNEL_SIZE` in the figure. Note that the messages inside the communication channels are of the form { `byte, byte` }, which corresponds to the tuple (x, v), where x is the

variable, v the new value. Instead of sending the name of the variable, we use the distributed unique identifier of the variable.

The specification of `input_X` and `output_X` has to be written manually for each site `X` in the system. Besides the initialization (`init_site_X()`), which initializes some variables for internal housekeeping, the behavior of a site is exactly an infinite loop of Input-Process-Output cycles.

The `read_msg_X` part of the behavior is technically more tricky. It handles the message channel for site `X` as follows. First, observe that the length of the message queue is inspected when a new cycle starts. Recall the behavior given in the formal semantics, where $\diamond_i$ markers are inserted to limit the number of messages in each subchannel, and where the message treatment phase ends when all markers are in front of the queue. We do not use markers here, but use the length of the queues to ensure that messages arriving during message treatment are not handled before the next cycle. Additionally, the message treatment can be ended anytime, due to the `break` guard of the loop. The reception of a message can be done from any originating process (this is modeled by the `if` with `N_SITES` guards). Once the message is removed from the queue, it is passed to `assign_X`, which contains a switch on all the distributed identifiers for the global variables, and assigns `val` to the variable with identifier `var` (`assign_X` is not included in the skeleton). Finally, the call to `when_X()`, enforces that the triggering conditions of all `WHEN`s are checked, and their bodies are executed if necessary.

In addition to the treatment of $\sim$ variables, we also handle `SEQUENCE`s in `read_msg_X`. A global variable `seq_i` exists for each `SEQUENCE`, indicating on which site the `SEQUENCE` resides (cfr. ξ in the formal semantics). If it points to the current site, the `SEQUENCE` can be executed, and the call to `seq_i_X()` executes a single instruction of the `SEQUENCE`. We detail the insides of this inlined function next.

Verification of programs with `SEQUENCE`s

The inlined function `seq_i_X` contains the behavior of `SEQUENCE` i on site X. Its purpose is to execute a single instruction of the `SEQUENCE`. Following the formal semantics, it must also change the value of `seq_i`, which indicates the site on which the `SEQUENCE` is active (cfr ξ in the formal semantics). For the translation of the workload σ_i of the `SEQUENCE`, we introduce an additional global variable for each `SEQUENCE` called `PC_i`. This variable reflects the instructions remaining to be executed for `SEQUENCE` i. The translation of the global valuation μ is straightforward : all local variables in `SEQUENCE`s are declared as shared global variables.

```
chan chan_X_1 = [MAX_CHANNEL_SIZE] of { byte, byte, byte };
//...
chan chan_X_N_SITES = [MAX_CHANNEL_SIZE] of { byte, byte, byte };

inline input_X() {
   // environment dependent behavior goes here
} // Analogue for output_X()

inline when_X() {
   // These are calls to all WHENs on site X
   // to check for a raising edge
   when_X1(); when_X2(); ... when_Xn();
}

inline read_msg_X() {
   byte var, val;
   do :: atomic {
           if :: sz1 -> sz1--; chan_X_1 ? var, val;
              :: sz2 -> sz2--; chan_X_2 ? var, val;
                 // ...
              :: sz_N_SITES ->
                       sz_N_SITES--;
                       chan_X_N_SITES ? var, val;
           fi;
           assign_X (var, val);
           when_X();
         }
      :: atomic { seq_1 == X -> seq_1_X(); }
      :: /...
      :: atomic { seq_L == X -> seq_L_X(); }
      :: break;
   od
}

proctype site_X() {
    atomic { init_site_X(); }
    byte sz1, sz2, /*...*/, szN_SITES;
    do  :: atomic {
               d_step {
                   sz1 = len (chan_X_1);
                   sz2 = len (chan_X_2);
                   // ...
                   sz_N_SITES = len (chan_X_N_SITES);
               }
               input_X();        // Read inputs
               when_X();         // Treat events
           }
           read_msg_X();  // Treat X's message queue
           atomic {
               output_X();       // Write outputs
           }
   od
}
```

Figure 6.1: PROMELA skeleton for a site

A skeleton of the PROMELA specification for `seq_i` is given in figure 6.2. First the instruction is executed, then the next instruction is selected, by changing `PC_i`. There is a branch in the `if` for each instruction of the `SEQUENCE` executable on site `X`. Finally, the `SEQUENCE` is migrated to any site that is able to execute this next instruction. This is modeled by the nondeterministic `if` with all guards evaluating to 1.

```
inline seq_i_X() {
    if :: PC_i == j -> // execute j-th instruction
                       ... // Trigger WHENs, send updated values
                           // if necessary

                       // update the program counter
                       PC_i = next_instruction(j);

                       // nondeterministically change site
                       if :: 1 -> seq_i = site_1;
                          :: 1 -> seq_i = site_2;
                          // ...
                          :: 1 -> seq_i = site_l;
                       fi;
       :: // ...
    fi
}
```

Figure 6.2: PROMELA specification skeleton for `SEQUENCE`s.

For programs including `SEQUENCE`s, the backend produces a model that is much closer to the implementation. Indeed, since the compiler is used to produce the PROMELA code, all instructions are assigned to a certain site, using the coloring algorithm of section 4.3. Synchronization code is inserted, and messages are sent over the channels, as described in chapter 5. The set of possible behaviors described by this model is smaller than, but included in, the set of behaviors of the semantic model. The lattice of behaviors presented in chapter 3 does not hold for the models produced by the compiler-distributer's PROMELA backend.

The lack of the automatic translation of $_d$SL's semantics for `SEQUENCE`s is a historical fact. There are no unsuperable difficulties, and we hope that with the future development of our tool this feature will be integrated.

6.2.2 State space reduction and the use of `atomic`

We use PROMELA's `atomic` construct to merge all transitions in the process phase of a particular process together into one meta transition[1]. To justify this reduction, observe that exterior processes cannot have any effect on the behavior of the process phase and that its progression has no influence on other processes. Indeed, a process is only influenced by its FIFO channels and the environment, and neither of them are consulted during processing. On the other hand, the environment remains unchanged during this phase, and another process can only observe the modification of the FIFO channels when messages are sent. Although the introduction of `atomic` groups such messages together, this has no effect since the number of messages received in the input phase is non deterministic. Unfortunately, not the entire cycle can be placed inside an `atomic` block. Indeed, the semantics of `SEQUENCE`s is such that it can influence a process while it is in its message treatment phase.

Caused by our asynchronous execution scheme which imposes that processes can only observe their own local variables, this reduction can be applied to any $_{\mathrm{d}}$SL program. Notice that the process phase contains most of the controller's behavior, and that this reduction is therefore substantial.

6.3 Results

6.3.1 The canal lock controller

We now explain how to translate the $_{\mathrm{d}}$SL program for the canal lock controller into PROMELA. Remember that this translation depends on a given distribution. Here, we have considered several possible distributions, from the minimal distribution to the maximal one. For this particular application, the maximal distribution consists of 11 sites, where 4 of the 11 sites each monitor a single control button responsible for sending the command to close one of the gates. Results for this extreme configuration are hard to obtain since the number of processes makes the state space size explode. In the 7-sites distribution, the control panel, each gate and each water level is controlled by a single site, while the 3-sites distribution has the control panel on one site, and each lock controlled by another site. The 2-sites distribution has one site for both locks, and one for the control panel.

Note that each distribution with more sites is a refinement of a distri-

[1] Note that, since messages are sent during the process phase and FIFO channels might be full, the use of PROMELA's `d_step` is prohibited

```
inline flip_flop_behavior(sensor_flipped, sensor_flopped,
                          order_cmd, order_dir) {
   if :: order_cmd && sensor_flopped ->
         if :: order_dir == ORDER_TO_FLOP -> skip;
            :: order_dir == ORDER_TO_FLIP -> sensor_flopped = false;
         fi;
      :: order_cmd && !sensor_flopped && !sensor_flipped ->
         if :: order_dir == ORDER_TO_FLOP -> sensor_flopped = true;
            :: order_dir == ORDER_TO_FLIP -> sensor_flipped = true;
            :: skip;
         fi;
      :: order_cmd && sensor_flipped ->
         if :: order_dir == ORDER_TO_FLOP -> sensor_flipped = false;
            :: order_dir == ORDER_TO_FLIP -> skip;
         fi;
      :: skip;
   fi;
}
```

Figure 6.3: Flip flop behavior

bution with less sites, which allows us to illustrate the simulation relation used in theorem 2. Note however that, in practice, due to this theorem, the correctness of the 7-sites distribution (if feasible) is sufficient to prove the correctness of the 3-sites, 2-sites and 1-site distribution.

The gates and the water levels are modeled using a *flip-flop* behavior (see figure 6.3) that has four states : flipped (1,0), flopped(0,1), flipping(0,0) and flopping(0,0). It can receive an order to flip, to flop or to do nothing. The behavior is obvious, and can easily be adapted for the gates (flipped ≡ opened, flopped ≡ closed) as for the water level (flipped ≡ up, flopped ≡ down). A nondeterministic choice makes the gate (and the water) move from opened (up) to closed (down) by allowing the model to stay in the flipping, respectively flopping state. Modeling the operator is straightforward : a nondeterministic choice lets the operator choose one of the twelve buttons (`lock{1/2}.{top/btm}_gate.btn_{open/close}` and `lock{1/2}.btn_{empty/fill}`).

Problem in the locks controller! At first glance, this implementation seems to work. However, after modeling it in PROMELA as explained before and using *Spin* model checker to verify the given constraints, we found out that it is faulty. Indeed, as shown in figure 6.4, two consecutive gates (top gate of lock 1 and bottom gate of lock 2) can be opened at the same time. In this case, three orders are given to the top gate of lock 1: an order to open, followed by an order to close (before the gate is completely

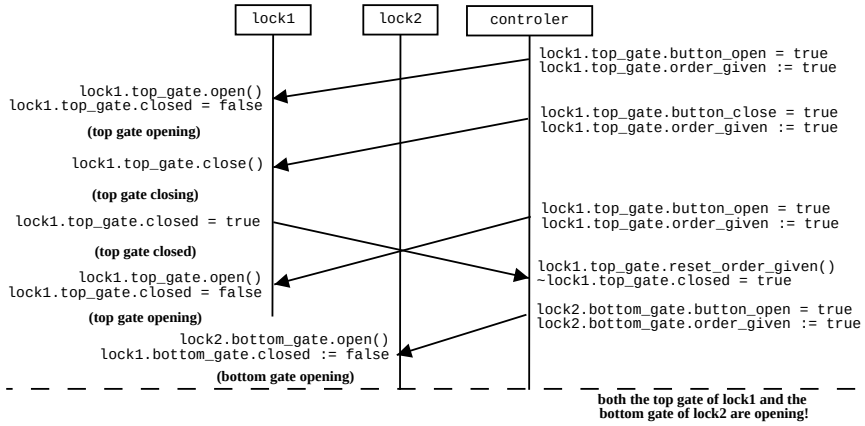

Figure 6.4: Error trace

opened) and finally an order to open. Because of communication, the `reset_order_given()` and the value of `~lock1.top_gate_closed` are delayed (respectively because of the `launch` and '~'). Therefore when the order to open the bottom gate of lock 2 is given, the controller believes that the top gate of lock 1 is closed and that no order has been given to it. This allows the opening of the bottom gate of lock 2, which violates the constraints.

An easy way to correct this, would be to allow a command to a gate (or a water level) only if its `order_given` is false (in other words, only allowing one order at a time). However, this would not be a viable solution. Indeed, imagine a boat gets stuck while the gate is closing, the controller would not allow to open a gate until it is completely closed, and the boat would be crushed! So, instead of blocking all commands while an order is processed, we disable the commands only during the time needed to verify the (distributed) constraints. To achieve this, a sequential execution checks that the issued command can be executed, by migrating the condition to all intervening sites. As illustrated in figure 6.5, this is done by means of a `SEQUENCE` that evaluates, in the local variable `check`, that all the conditions are satisfied. In the example of figure 6.5, the first part of the constraint (`check := (lock2.top_gate.closed and lock2.water_down);`) will be evaluated on the site where `lock2` is localized, then the value of `check` will be migrated to the site where `lock1` is localized to evaluate the second part (`check := (check and lock1.top_gate.closed);`). Since the control panel is disabled during this task, we can be sure that the variable `check` is true if and only if the constraints are satisfied, in which case, the corresponding action(s) is (are) taken.

```
WHEN lock2.bottom_gate.btn_open and not disabled THEN
  disabled := true;
  launch open_bottom_gate_lock2;
END_WHEN

SEQUENCE open_bottom_gate_lock2()
VAR
  check : bool;
END_VAR
  check := (lock2.top_gate.closed AND lock2.water_down);
  check := (check AND lock1.top_gate.closed);
  IF check THEN
    not_allowed_led := false;
    LAUNCH lock2.bottom_gate<-open();
  ELSE
    not_allowed_led := true;
  END_IF;
  disabled := false;
END_SEQUENCE
```

Figure 6.5: `WHEN` / `SEQUENCE` monitoring the command "open the bottom gate of lock2"

Verification Two properties were checked on the model. the first property expresses all the constraints imposed on the system. It is expressed in LTL using the formula `[] !global_bad` where `global_bad` is defined in *Spin* as follows:

```
#define global_bad (
    (!lock1_top_gate_closed && !lock1_bottom_gate_closed) ||
    (!lock1_top_gate_closed && !lock2_bottom_gate_closed) ||
    (!lock2_bottom_gate_closed && !lock2_top_gate_closed) ||
    (!lock1_bottom_gate_closed && !lock1_water_down)      ||
    (!lock1_top_gate_closed && !lock1_water_up)           ||
    (!lock2_bottom_gate_closed && !lock2_water_down)      ||
    (!lock2_top_gate_closed && !lock2_water_up)           ||
)
```

The second property expresses only the fact that the two middle gates (i.e top gate of lock1 and bottom gate of lock 2) of the locks are opened at the same time. It is expressed in LTL using the formula `[] !middle_bad` where `middle_bad` is defined in *Spin* as follows:

```
#define middle_bad (
    !lock1_top_gate_closed && !lock2_bottom_gate_closed
)
```

The verification was made using *Spin* version 4.2.4, on a 3 GHz Intel Xeon machine with 4GB of memory. Some representative results corre-

sponding to the verification of these properties on the first (faulty) version of the canal lock controller are shown in figure 6.6. The first column indicates the number of execution sites, respectively 1 (minimal), 2, 3, 7 and finally 11 (maximal). The second column indicates the maximum number of times each button can be pressed. The next column indicates whether or not messages are taken from their queues as soon as possible. The fourth and fifth columns show respectively the property that was checked, and whether or not partial order reduction was used. The four remaining columns show respectively, whether the property was verified or not, the time needed for the verification, the number of states explored and the memory usage. First of all, by examining the first two sets of experiments, we can observe that the results are coherent with theorem 2. That is, if the error in figure 6.4 is found in a distribution (indicated by $\times$ in column 6), it is also found in more refined distributions. Next, note that if the property is verified with a more refined distribution (indicated by $\surd$), it is also correct for less refined distributions. Finally, in the third set of experiments, we can observe that the gain due to partial order reduction on time and used memory is limited. The gain is substantial only for the maximal distribution. This is because, in the specification, we intensively use `atomic` blocks to reduce the state space, therefore reducing the efficiency of the partial order reduction.

Sites	Btn	Channels	Property	P.O.	Verif	Time	States	Memory
1	2	Instant	`global`	yes	$\surd$	0:03.25	1.41e+04	1.432 MB
2	2	Instant	`global`	yes	$\times$	0:14.52	8.37e+04	3.064 MB
3	2	Instant	`global`	yes	$\times$	0:08.00	9.54e+04	3.569 MB
7	2	Instant	`global`	yes	$\times$	0:09.74	7.82e+04	3.645 MB
11	2	Instant	`global`	yes	$\times$	5:38.48	1.41e+06	68.56 MB
1	1	Normal	`middle`	yes	$\surd$	0:07.80	3.98e+04	2.457 MB
2	1	Normal	`middle`	yes	$\surd$	0:43.22	1.29e+05	4.908 MB
3	1	Normal	`middle`	yes	$\surd$	0:37.13	1.39e+05	5.311 MB
7	1	Normal	`middle`	yes	$\times$	21:14.36	1.91e+06	77.581 MB
11	1	Normal	`middle`	yes	$\times$	0:54.93	2.07e+05	10.383 MB
1	1	Instant	`middle`	no	$\surd$	0:00.15	437	0.409 MB
1	1	Instant	`middle`	yes	$\surd$	0:00.15	437	0.409 MB
2	1	Instant	`middle`	no	$\surd$	0:01.04	8133	0.709 MB
2	1	Instant	`middle`	yes	$\surd$	0:01.08	8132	0.709 MB
3	1	Instant	`middle`	no	$\surd$	0:00.72	10136	0.805 MB
3	1	Instant	`middle`	yes	$\surd$	0:00.73	10133	0.805 MB
7	1	Instant	`middle`	no	$\surd$	0:11.46	146401	6.206 MB
7	1	Instant	`middle`	yes	$\surd$	0:11.96	146338	6.206 MB
11	1	Instant	`middle`	no	N/A	*	2.10+07	1 GB
11	1	Instant	`middle`	yes	$\surd$	18:03.57	5.43e+06	261.7 MB

Figure 6.6: Results of the verification of `global_bad` and `middle_bad` on the first (faulty) version of the canal lock controller.

6.3.2 The conveyor belt

Consider the conveyor belt system presented in section 3.7.2. We show here how the system can formally be verified with *Spin*, using the compiler-distributer's PROMELA backend.

Manual intervention Several modifications had to be done to make the dSL program verifiable with *Spin*.

First of all, there is a cyclic triggering of `WHEN`s which cannot be translated by the PROMELA backend of the compiler-distributer, since PROMELA does not have a straightforward way of handling recursive calls. These `WHEN`s are as follows :

```
WHEN ~belt_in1.box_at_end
     AND mobile_hor_free
THEN
     mobile_hor_free := FALSE;
     LAUNCH belt_in1_to_lower_central();
END_WHEN

WHEN ~belt_in2.box_at_end
     AND mobile_hor_free
THEN
     mobile_hor_free := FALSE;
     LAUNCH belt_in2_to_lower_central();
END_WHEN
```

Indeed, the assignment to `mobile_hor_free` could cause cyclic triggering. Without a semantic analysis, the PROMELA backend cannot decide that there is no problem with this code. Note that the assignment indeed does not cause problematic behavior since the triggering conditions of all intervening `WHEN`s evaluates to false. The removal of the cyclic triggering has therefore to be made by hand in the produced PROMELA code. In practice, the PROMELA backend produces a *call* to an `inline` function, which computes the condition of the `WHEN`, and triggers its body if necessary. This *call* has to be removed and replaced by an assignment to the variable storing the old condition of that `WHEN` (which evaluates to `FALSE`).

A more elaborate manipulation is needed for the suppression of the `WAIT` instructions inside the `SEQUENCE`s, since those are not supported by the PROMELA backend. We illustrate this with the beginning of `belt_in1_to_lower_central()`, which is changed according to the translation presented in section 3.6, as follows :

```
SEQUENCE belt_in1_to_lower_central_part1()
    mobile_hor<-go1();

    _belt_in1_to_lower_central_part1.WAITING := TRUE;
END_SEQUENCE

WHEN _belt_in1_to_lower_central_part1.WAITING AND
     mobile_hor.pos1 THEN
     _belt_in1_to_lower_central_part1.WAITING := FALSE;
     LAUNCH belt_in1_to_lower_central_part2();
END_WHEN

SEQUENCE belt_in1_to_lower_central_part2()

    // Waited until the mobile horizontal belt
    // is in the right position

    // ...
END_SEQUENCE
```

Remark that this translation requires the introduction of a fresh variable for each suppressed `WAIT`.

Choice of distribution The distribution used for the verification of this system consists of 6 sites, each sites governs a given conveyor belt. The two sites each connected to a mobile belt have 3 inputs (`belt.pos1, belt.pos2` and `belt.box_at_end`) and two outputs (`motor` and `belt.motor`) assigned to it. The sites controlling a fixed belt govern one input (`box_at_end`) and one output (`motor`).

Modeling the environment The environment is modeled using two `inline` PROMELA constructs : one describing the behavior of a fixed belt, and one for a mobile belt.

```
inline mobile_belt_behavior(motor, pos1, pos2) {
    if :: motor > 0 && pos1  -> pos1 = 0; // Going to pos2
       :: motor > 0 && !pos1 -> pos2 = 1; // In pos2 now
       :: motor < 0 && pos2  -> pos2 = 0; // Going to pos1
       :: motor < 0 && !pos2 -> pos1 = 1; // In pos1 now
       :: skip; // Nondeterministically let time pass
    fi
}

inline fixed_belt_behavior(motor, belt_box_at_end, box,
```

```
                          next_box, next_is_running) {
    if :: motor && !belt_box_at_end && box
          // Motor is running, a box is on the belt
          // make it appear on the end of the belt
          -> belt_box_at_end = 1;
       :: motor && belt_box_at_end
          // Motor is running, a box is on the end of
          // the belt. Make it go onto the next belt
          -> belt_box_at_end = box = 0; next_box = 1;

          // Check constraint
          assert (next_is_running);
       :: skip; // Nondeterministically let time pass
    fi
}
```

To keep track of the boxes in the system, six global boolean variables (`box_on1` through `box_on6`) are added in the PROMELA specification of the environment. Each such a variable will be set to `true` when a box is on the associated belt. Using these variables and the previous two pieces of PROMELA code, the environment's behavior is easy to specify :

```
inline input_1() { // Behavior of belt in1
    if :: !box_on1 -> box_on1 = 1; // Box created on belt in1
       :: skip; // Non deterministically do nothing
    fi;

    // Check constraints
    if :: belt_in1_motor && belt_in1_box_at_end ->
          assert (mobile_hor_pos1);
       :: else; // To avoid deadlock
    fi;

    fixed_belt_behavior(belt_in1_motor,
                        belt_in1_box_at_end,
                        box_on1, box_on3, mobile_hor_belt_motor);

} // Equivalent for input_2() modeling belt in2

inline input_3() { // Behavior of mobile_horizontal belt
    mobile_belt_behavior(mobile_hor_motor,
                         mobile_hor_pos1, mobile_hor_pos2);
    fixed_belt_behavior(mobile_hor_belt_motor,
                        mobile_hor_belt_box_at_end,
                        box_on3, box_on4, belt_central_lower_motor);
```

```
}
// ...
```

Notice that the safety constraints for this system (a box can only leave a belt when the next belt is in front of it and is moving) are incorporated in the description of the environment by means of PROMELA's `assert` instructions. When an exhaustive traversal of the state-space is performed with *Spin*, these `assert` instructions are checked for all applicable states and an error is reported with a counter-example if a condition in an `assert` instruction is not verified.

Results The 6 sites distribution of the conveyor system respects the previously mentioned constraints imposed on the system. The above PROMELA specification can be checked in a reasonable amount of time using *Spin*. *Spin*'s output can be found in figure 6.7. Execution times are obtained on a 3 GHz Intel Xeon machine running Linux with 4GB of memory. The results are obtained using 2.95 GB of memory, and less than 6 minutes of CPU time. The complete state space consists of $4.79 \cdot 10^7$ states.

```
Depth=    3060 States= 4.78553e+07 Transitions= 5.1146e+07 Memory= 2945.991
(Spin Version 4.2.4 -- 14 February 2005)
        + Using Breadth-First Search
        + Partial Order Reduction
        + Compression

State-vector 256 byte, depth reached 3062, errors: 0
4.78553e+07 states, stored
          425294 nominal states (stored-atomic)
3.29074e+06 states, matched
5.11461e+07 transitions (= stored+matched)
4.743e+07 atomic steps
hash conflicts: 2.91227e+06 (resolved)

Stats on memory usage (in Megabytes):
12825.224       equivalent memory usage for states
2677.875        actual memory usage for states (compression: 20.88%)
2945.991        total actual memory usage

341.07user 4.19system 5:45.32elapsed
```

Figure 6.7: *Spin*'s output for the Conveyor system model

The model could however not be verified using normal depth first search, since verification times using this standard exploration appear to be very long (the verification was halted after 3 days, with only $7 \cdot 10^5$ states explored). Breath first search had to be used to get the results. A quick

investigation leads us to suspect that the state space has a particularly *deep* structure (the length of the search stack in depth first search is over $5 \cdot 10^5$ states long). This *deep* structure is combined with an important number of transitions that go back to states that are already on the depth first search stack. This results in an inefficient exploration, since these transitions have to be undone (backtracked). Since a lot of transitions are inside the `atomic` construct, many of them have to be calculated and backtracked, which worsens the exploration time.

6.3.3 The railway system

Manual intervention In order to check the railway system, using the PROMELA backend of the $_d$SL compiler-distributer, a small change to the produced PROMELA code had to be made. The problem is caused by recursive `WHEN`s and is analogue to the one encountered in the conveyor belt example:

```
WHEN cs.waiting_1 AND cs.flag == 0 THEN // WHEN 1
    cs.waiting_1 := FALSE;
    cs.flag := 1;
END_WHEN
WHEN cs.waiting_2 AND cs.flag == 0 THEN // WHEN 2
    cs.waiting_2 := FALSE;
    cs.flag := 2;
END_WHEN
```

Note that the assignment to the `waiting` variables does not introduce problematic (recursive) `WHEN` triggerings, and can therefore safely be modified by hand.

Chosen distribution The verification results of the railway system are performed using the following 4 site-distribution

Site	Variables
1	`out_1`, `cs_flag`, `cs_waiting_{1,2}`
2	`out_2`
3	`in_1`, `train_1.motor`
4	`in_2`, `train_2.motor`

Modeling the environment To model the environment, i.e. both trains, two additional boolean variables are used : `train_1_in_cs` and `train_2_in_cs`. Each of these variables decides whether or not a train is in its critical section. The specification of the environment is as follows :

```
inline env_1() {
   if :: out_1 && train_1_motor ->
         // When the train is before the out_1 sensor
         // And it is running, make it leave the sensor
         out_1 = 0;

      :: else -> if :: train_1_in_cs && train_1_motor ->
                       // When the train is in its critical section
                       // and is running, make it pass before
                       // the out_1 sensor, and leave the critical
                       // section
                       out_1 = 1; train_1_in_cs = 0;
                    :: skip;
                       // Let time pass
                 fi
   fi
}

inline env_3() {
   if :: train_1_motor && ! train_1_in_cs ->
         // Make the train go in front of the in_1 sensor
         // (i.e. the train approaches the critical section
         in_1 = 1;
      :: in_1 && train_1_motor ->
         // If the train approached the critical section,
         // and is moving, make it enter the critical section
         in_1  = 0; train_1_in_cs = 1;
      :: skip;
         // Let time pass
   fi
}
```

The behavior for `env_2()` and `env_4()` is analogue.

Results The railway system is correct: it is not possible that both trains are in the critical section at the same time. In order to automatically verify this constraint with the previously described model, we use PROMELA's `never` claim. The `never` claim can be seen as an additional process which monitors the system. LTL formulae can automatically be transformed into `never` claims by the *Spin* tool. The never claim corresponding to the critical section property is as follows :

```
never {
  do :: (train_1_in_cs) && (train_2_in_cs) -> break;
     :: else;
  od;
```

```
  // When the end of a never claim is reached,
  // an error occurs
}
```

The results of the verification are presented in figure 6.8. The verification is performed on a 3 GHz Intel Xeon machine running Linux with 4GB of memory. The verification takes less than 30 seconds, and uses not more than 60MB of memory. Note that most of the memory is needed for the depth first search stack (38MB). As is the case in the previous example, the statespace has a very *deep* structure. The total size of the state space is relatively small, counting $5.07 \cdot 10^5$ states. Results with breath first search are comparable to the ones obtained by using depth first search.

```
State-vector 172 byte, depth reached 1037646, errors: 0
  506853 states, stored
5.86972e+06 states, matched
6.37657e+06 transitions (= stored+matched)
6.871e+07 atomic steps
hash conflicts: 1.40206e+06 (resolved)

Stats on memory usage (in Megabytes):
93.261  equivalent memory usage for states (stored*(State-vector + overhead))
17.854  actual memory usage for states (compression: 19.14%)
2.097   memory used for hash table (-w19)
38.400  memory used for DFS stack (-m1200000)
58.315  total actual memory usage

27.00user 0.09system 0:27.10elapsed
```

Figure 6.8: Results of *Spin*'s verification of the railway system

Chapter 7

Discussion and future work

In this chapter, we discuss the open questions concerning $_{\mathrm{d}}$SL. We first comment the features present in SL, but missing in $_{\mathrm{d}}$SL. All of these features are hard to implement due to the static distribution used in $_{\mathrm{d}}$SL. These features are discussed in section 7.1.

Next, in section 7.2, we give some hints on how the $_{\mathrm{d}}$SL semantics (cfr. chapter 3) could be extended to include real-time. Indeed, the semantics of $_{\mathrm{d}}$SL is currently untimed, and real-time semantics could be interesting since they give a more detailed and realistic description of systems implemented with $_{\mathrm{d}}$SL.

In section 7.3, we discuss future work concerning the distribution algorithms of $_{\mathrm{d}}$SL (cfr. chapter 4). Both algorithms, the shrinkage algorithm from section 4.3 and the instruction reordering algorithm from section 4.5 are subject to further study.

Finally, in section 7.4, we discuss the verification of $_{\mathrm{d}}$SL, presented in chapter 6. This verification is valuable for assuring the correctness of the system. Unfortunately, the state space explosion problem makes it hard to obtain results for large systems when an exhaustive search of the state space is performed. We give therefore some alternatives to increase the confidence in the correctness of a system implemented with $_{\mathrm{d}}$SL. These alternatives do not necessarily perform a complete exploration of the state space.

7.1 Missing features in $_{\mathrm{d}}$SL

The fact that the localization of instructions and global variables is decided at compile time has a major impact on several features of SL. As a result of this choice, all dynamic elements of the language SL are hard to implement in $_{\mathrm{d}}$SL. These elements include pointers, arrays, polymorphism and

inheritance. These problems are addressed in section 7.1.1.

Next to those language elements, the static distribution makes modular compilation of dSL programs impossible, which has a drawback on the maintenance of large projects and code reusability. We look into these matters in section 7.1.2.

7.1.1 Dynamic elements in dSL

Problems

SL is an object oriented language including all the fundamental concepts of an OO language : objects, abstraction, encapsulation, polymorphism and inheritance. However, dSL's static distribution makes these concepts hard to implement.

The static distribution used in dSL requires all global variables and instructions to be located on a certain site at compile time. Moreover, method calls have to be resolved at compile time, which means that when a method is called, the compiler has to know the site on which that method resides. With SL's inheritance, the instructions that are executed when a method is called depends on the object that is used to make the call. For example, when a hydraulic pump is called to start, it will initialize its pressure, while the same action for an electric pump may involve initializing currents and voltages. This means that the call to the method `start` on an object of type `pump` may lead to the execution of different codes, which may reside on different sites.

Even if it would be possible to find out the type of object that is called at compile time to know what instruction to call, this information would not suffice to solve the problem. Indeed, consider for example two objects `pump1` and `pump2` of type `pump` having their attributes localized on different sites. Suppose further that a method `start` is defined for objects of type `pump`, initializing the `pump`'s attribute `current`. Supposing that this attribute is an external variable, it might be localized on different sites depending on the particular instances of the type `pump`. Obviously, when a call to `pump1<-start()` or `pump2<-start()` is made, there is no problem for the compiler to know which code is called. However, if a pointer `pump_ptr` is used to make the call `pump_ptr<-start()`, the compiler needs to know what instance of `pump` is used to produce correct code. Solving this problem is obviously undecidable in general. An analogue problem, to the one caused by the use of pointers, occurs when arrays are used.

The need for pointers is substantial since the lack of them limits code reuse. A typical situation in which pointers could come in handy arises when

one object, say a `pump`, is *part of* another larger object, for instance a `lock`. Often, the child object wants to invoke methods on its parent object, which is easy to achieve using pointers.

Without pointers, the user either has to duplicate the code, or can use arrays instead. In the latter case, all global objects of the same type are put into a global array. In our `lock` and `pump` example, this would mean that all `lock`s are in an array called `lockarray`. Then, instead of using a pointer `lock_ptr` to a `lock`, the `pump` can use an integer `lock_idx`, which is the index of that `lock` in the `lockarray`. The call to `lock_ptr<-M()` is then equivalently replaced by `lockarray[lock_idx]<-M()`.

Possible solutions

A possible solution to the above described problems consists in generalizing the specialization approach of section 5.1.3. Notice that with specialization, array access is removed from the code. This approach can be used to remove pointers as well.

Remark that the solution for arrays is simplified by the fact that the number of objects to which an array access refers is statically known. While this solution could be applied to pointers, the situation is more complex. The compiler-distributer would need to have a list of possible objects to which a pointer points at a given location in the program. Using this list, `METHOD`s could be specialized and code transformed as explained for arrays. Techniques that try to find for each pointer and each program location an as small as possible set of objects to which a pointer may point are known in the literature as points-to-analysis techniques [HP00].

Although the problem of finding the exact set of possible objects is undecidable in general, many approximating algorithms exist. The most naive conservative approximation supposes that each pointer may point to all objects of compatible types (including subtypes). More elaborate algorithms which implement points-to-analysis range from very simple [IH97] to expensive and more accurate [EGH94] approximations.

Remark however that in $_d$SL, points-to-analysis is simpler than the general case considered in the above cited work. Indeed, $_d$SL does not include dynamic allocation, and does not allow pointer arithmetic.

Atomic code in the presence of pointers

Consider the following snippet of code :

```
CLASS Engine
   current   : INT;
END_CLASS

CLASS Pump
   ptr_engine : POINTER_TO Engine;
   temp       : INT;
   valve      : BOOL;
END_CLASS

GLOBAL_VAR
   pump1, pump2     : Pump;
   engine1, engine2 : Engine;
   ptr_pump         : POINTER_TO Pump;
END_VAR

WHEN pump1.temp > 80 THEN
   ptr_pump := ADDRESS OF pump1;
   pump1.ptr_engine<-stop();
END_WHEN

WHEN pump2.temp > 80 THEN
   ptr_pump := ADDRESS OF pump2;
   pump2.ptr_engine<-stop();
END_WHEN

METHOD Engine::stop()
   current := 0;
   ptr_pump.valve := FALSE;
END_METHOD

// ...
```

In this example, the first `WHEN` forces `pump1.temp` and `pump1.ptr_engine` on the same site. But what about `engine1.current` and `engine2.current`? The call to `stop` on `pump1.ptr_engine` is ambiguous. If `pump1.ptr_engine` only points to `engine1`, then `engine1.current` and `engine2.current` can be localized on different sites. If it may point to both engines, both `current` variables must be localized on the same site as `pump1.temp`. Although we cannot answer this question in general, points-to-analysis can be used as an approximation. The program remains distributable with both pumps on different sites if it turns out that `pump1.ptr_engine` points to `engine1` and `pump2.ptr_engine` to `engine2`. In that case, the method `stop` will be duplicated and will exist in two versions : one for each engine.

It is however not clear if the distributability of a program should depend or not on the efficiency of the points-to-analysis. We think it should not, and more work is therefore needed to define clearly how atomic code in the presence of pointers (cfr. definition 20 of synchronous flow) affects the distributability of a $_d$SL program.

7.1.2 Separate compilation

Separate compilation is a desired functionality which enables modularity and code reuse [GBJL02]. With separate compilation, code can be precompiled and organized into libraries, each library providing a number of functionalities which can be reused in different applications.

In a typical imperative language, the code inside a library has gone through all compilation phases including parsing, syntax and semantic checking, transformation into control flow graphs, generation of intermediate code, optimization and finally generation of executable code. When the precompiled code is used in conjunction with a program, only very small changes have to be made to the code inside the library. A linker performs this job, which consists in relocating addresses of global variables and code in memory in such a way that variables and code residing in the library can be accessed by the program [PW72].

The benefit of using libraries is twofold. On the one hand, a library can be used by a program which only has to know the interface, i.e. the functionalities provided by the library, without knowing the internal implementation details of the code inside the library. On the other hand, there is a reduction in the compilation time : since the code inside a library has gone through all compilation phases, the time spent by the compiler can be replaced by the time needed by the linker. Since linking is far less expensive than compiling, the compilation time of a program in conjunction with a library is far inferior than the time needed compiling both as an indivisible whole.

Both advantages are non existing in $_d$SL in general. The modularity advantage is broken by the atomic constraints which are introduced when a program makes synchronous calls to code which would be in a library. These calls may introduce additional constraints between variables inside the library, making the conjunction of the program and the library non distributable. When this happens, the programmer is faced with the internals of the library, which may involve its implementation details. The advantage of having small linking times compared to compilation times is also non existing in $_d$SL. Indeed, the localization of variables in $_d$SL has an impact

on the executable code inside a library. The localization of variables which would reside in the library are dependent on how the library is used. This results in the fact that the precompiled code would have to undergo substantial changes depending on how the code is used. These changes include the synchronization code inside dSL's `SEQUENCE`s (section 5.1.4), sites to contact when code is `LAUNCH`ed (section 5.1.5) and duplication of code (section 5.1.3). These changes, in contrast to the knowledge required by a common linker, are fundamental and can only be performed with deep knowledge of the code to compile.

7.2 A real time semantics for dSL

In [Mic05], the author performs a preliminary study on how the dSL semantics can be extended to include real-time.

The first step in this approach consists in the extension of dSL's syntax, by adding temporal keywords, including *duration* keywords which measure elapsed time, and *time of day* keywords which allow to express an absolute time reference.

Next, the semantics is adapted to include real-time. The semantics is no longer expressed as a labeled transition system. Instead, it is expressed as a network of timed automata. Timed automata are introduced by Rajeev Alur and L. Dill in [AD90, AD94] and are a special case of linear hybrid systems [ACH+95]. Three aspects of time are modeled using these timed automata: *transmission time*, *instruction time* and *current time*.

Transmission time models the time needed for messages to travel over the network. This is modeled by adding a timestamp to each message, and changing the message treatment rule by accepting only messages who's timestamp is in a certain interval of time.

The instruction time models the speed of the processor executing the dSL instructions. The principle is similar to that of a timeout clock. A clock is added to every process, which is reset on each transition described by the semantics. This clock is used to express that the following transition can only happen when the clock is within a certain interval of time.

The current time models the real world time, and is global to all processes. It allows the dSL program to trigger events when a certain global time is reached.

We refer the reader to [Mic05] for further reading, as well as the preliminary results on the formal verification of real-time dSL systems.

7.3 Future work on the distribution algorithms

Two algorithms are presented in chapter 4. The first algorithm is reduced to the NP-Complete multiterminal cut problem (cfr section 4.3), and we generalized theorem 5 due to Dahlhaus in such a way that it handles more cases, resulting in theorem 6. Two interesting questions remain.

The first question is how our shrinkage theorem could be applied into a branch and bound context [LW66]. The idea is the following : (1) shrink the graph (2) branch: select a remaining edge, contract that edge and shrink the graph (3) bound: Use a linear relaxation as a lower bound. Using this scheme, the shrinkage theorem could be used to reduce the state space in a branch and bound search. For the linear relaxation, one could use the formulation presented in [CKR00] or [KKS$^+$99].

The best approximation algorithm known is the one from [KKS$^+$99]. This algorithm is based on a linear relaxation of the $\{0, 1\}$ formulation of the multiterminal cut problem. A rounding procedure is used to come up with an integral solution, which is proved to be within 1.3438 of the optimal solution. Without going into the details of this paper, we can state that the rounding procedure is central in their approach. This rounding procedure divides a set of nodes into two subsets, each resulting set ending up into a different partition in the final solution. It would be interesting to study if their rounding procedure is sensitive to our shrinkage theorem. In other words : could their rounding procedure end up separating two nodes for which we are sure that there is an optimal solution which keeps them in the same partition ? If this is the case, then we can improve their algorithm, if this is not the case, then it would be interesting to relate our result with theirs.

Another point that could be interesting to study with the shrinkage algorithm is how the MAX-MIN-`st` cuts relate to each other in the different iterations of our unshackling heuristics. More precisely, suppose that MAX-MIN-`st` $\text{cut}_G(n, n')$ is known for two nodes n, n' in G. Now suppose that G is transformed into G' by contraction of a certain edge e in G. We have currently no knowledge of a relation between MAX-MIN-`st` $\text{cut}_G(n, n')$, MAX-MIN-`st` $\text{cut}_{G'}(n, n')$ and e. It would be interesting to study if there is a relation between these elements, and if such a relation exists, how it could be exploited to speed up our unshackling heuristics.

The other algorithm presented in chapter 4 is used to reorder sequential instructions in order to increase the performance of a $_{\text{d}}$SL program during execution. This problem is related (cfr. section 4.5) to the Precedence Con-

strained Class Sequencing problem (PCCS). We could further study both the theoretical and the practical point of view of our heuristics. From the practical point of view, with this problem, it is hard to measure the quality of a heuristics since a realistic set of instances is hard to obtain. Remember that we use *def-use* chains as the input for the precedence constraints. To the best of our knowledge, no statistical information exists on the presence of these chains in common programming languages. Using this information, we could finetune our heuristics, and compare them better with existing ones. From the theoretical point of view, the heuristics presented are hard to grasp. Their behavior is hard to generalize, and we therefore have not found hard instances, nor were we able to prove the existence of an approximation bound.

7.4 Partial verification and testing

In the previous chapter, we presented the verification of $_{\mathrm{d}}$SL. However, even for small examples, the state space explosion problem was already problematic. In this section, we present new ideas on how to handle the state space explosion problem.

In the literature, several solutions are proposed. One idea concentrates on building a *sound* abstraction of the system, either manually or automatically. A sound abstraction aims at finding a compromise between functional preciseness and functional correctness. An abstracted model of the system has a less detailed description of the system's behavior, and may therefore include behaviors that do not occur in the reality. However, if the abstraction is *sound* and no errors are found, then one can automatically conclude that no errors are present in the detailed description of the system, since the abstraction includes all behaviors of the complete system. On the other hand, due to the abstraction, the verification may come up with errors in the abstract model which are not present in the real system.

In what follows, we will not use the concept of abstraction, instead we introduce the notion of partial verification. Partial verification aims at exploring those parts of the state space that are *interesting*. The interesting parts of the state space should be defined in such a way that when these parts are explored, one can gain confidence in the correctness of the system. It is clear that if we do not perform an exhaustive exploration of the state space, there is no guarantee that when no errors are found, the system is correct.

We introduce two options for partial verification, which are based upon two different interpretations of what we call interesting parts of the state

space.

Related work

The work presented here is related to *heuristic* or *directed* model checking, a technique used to quickly find errors in large state spaces. In that work, an evaluation function is used that guides the model checker into the direction of bad states, resulting in shorter error traces.

Many of such heuristics are tailored to find a certain kind of error (e.g. [LCL88], [YD98]) while others use the structure of the property or the model to direct the model checker (e.g. [ELL01b], [GV02]).

Closely related to this work, is the technique presented in [ELL01b] and [ELL01a]. The authors propose an extension to SPIN for the heuristic exploration of large state spaces. As is the case here, the authors use the model checker as a debugging aid to find residual design and code faults. Using A^* [HNR68], an estimation of the distance from a given state to a bad state and the structure of the property, the model checker's state space exploration is guided in order to find short counter-examples. In [ELL01c], very similar techniques are used in order to obtain short counterexamples given a long trace to an error state.

Another ongoing area of research is to use the model checker's ability to find counter-examples in order to produce test cases [ADG+03]. Related to this work is the use of verification techniques to establish the conformance of the implementation with the formal specification by deriving test cases from the specification [JCTG96]. This can also be done by stimulating the implementation and observing its behavior [FJJV97].

In [GV02], the structure of the program is exploited to explore *interesting* parts of the state space. In this particular paper, branch coverage is used to direct the model checker : all high level branches are covered during exploration. This work is directly related to the test coverage guided verification presented in the next section.

Test coverage guided verification

With test coverage guided verification, we try to gain confidence in the software which is being verified by looking in those parts of the state space that are needed to satisfy some test coverage criterion. The complete description is used as the specification, meaning that exhaustive verification can not be performed a priori. Next, a test coverage metric is picked which will be used to guide the model through the state space. When several new states have

to be explored, the model checker chooses that state that most likely gives an increase in the chosen test coverage metric.

In this way, the model checker is used as an automatic testing tool. If the complete state space can be searched in a reasonable amount of time and memory, the model checker can perform an exhaustive state space search, and correctness can be proved. If this is not the case, the model checker terminates prematurely due to lack of memory or time, the test metric will indicate how much has been explored and confidence in the system may be increased. Notice that upon premature termination, the guided model checker will terminate with a higher coverage for the chosen metric than the standard model checker performing depth first search.

The most widely accepted test metrics [Pat05, Nta88] that can be used to guide the model checker are the following :

- Statement coverage, this measure reports whether each executable statement is encountered
- Decision coverage, this measure reports whether boolean expressions tested in control structures evaluated to both true and false.
- Condition coverage, this coverage reports the true and false outcome of each boolean sub-expression.
- Multiple condition coverage, reports whether every possible combination of the outcome of boolean sub-expressions occurs.
- Modified Condition/Decision Coverage (MC/DC) requires enough test cases to verify every condition can affect the result of its encompassing decision[CM94]. More specifically, every decision in the program must have taken all possible outcomes at least once, and each condition in a decision must been shown to independently affect the decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding all fixed all other possible conditions. For a formal definition, see [VB02]. This measure is required for aviation software [RCT92].
- Path coverage, which reports whether each of the possible paths in each function have been followed

Guiding the model checker to satisfy any of these metrics is not easy. Basically, the model checker's depth first search should be changed into a more general search method. For safety properties, the model checker could be changed as follows : using a priority queue, the model checker takes the

node with the highest priority from the queue. Initially, only the initial state is in the queue. When a node is taken from the queue, the model checker verifies the safety condition, and expands all successor nodes, for which it calculates the heuristics value, and puts them into the priority queue with their respective heuristics value.

Two interesting questions can be investigated in an in depth study of this approach. First, there is the question of finding a good heuristics function. Next, suppose that a good heuristic function is used, then it remains to see to what degree a certain test coverage metric is able to uncover errors. Note that with this approach the model checker is guided to increase the test coverage metric, in contrast to a guided search in the direction of the error. The strength of this approach relies on the strength of the test coverage metric. It would be interesting to use this method on faulty models, and study how efficient a certain test coverage metric is at finding errors.

We give here a simple approach, based upon the ideas of [GV02], which could be used to find a good heuristic value in order to make the model checker go into the right direction. In that paper, the authors define a set of distances between states in the state space. We will use two of the distances defined: the most simple distance metric is the hamming distance [Ham50] between the binary representation of two states s, s', let it be denoted by $H(s, s')$. The second distance is defined as the local control location distance, which is the shortest path inside a single process between two control locations l, l' (i.e. the smallest number of transitions one must fire to go from one control location to the other), let it be noted $D(l, l')$.

Now consider the MC/DC coverage metric which is the most general test coverage metric presented above. Suppose that the set M of tuples (c, v, l, p) is given, where c is a sub-expressions that must take value v to satisfy the MC/DC coverage metric, l is the control location of process p in which c is used[1]. A heuristic value $h(s)$ for a given state s could be calculated as follows :

$$\sum_{(c,v,l,p)\in M} D(s_p.loc, l) \cdot H(s(c), v)$$

where $s_p.loc$ is the current control location of process p and $s(c)$ is the value of c in s. Notice that high priority means lowest values with this heuristic function.

This heuristic search method has been plugged into *Spin*. Very preliminary studies have revealed the insufficiency of the MC/DC criterion. Indeed, its definition does not involve concurrency, and when this heuristic is used

[1] Multiple tuples with different p and l can be used if necessary

on faulty models, the model checker almost always reaches 100% coverage *before* finding the error. This means that even when 100% MC/DC coverage is achieved using a certain number of test cases, the errors where not uncovered. Otherwise stated, the MC/DC coverage has little impact on the confidence one can have in a certain system.

More work needs to be done to confirm these findings. If this observation would be confirmed, one should define better test coverage metrics, especially if concurrent systems are involved.

Trace guided verification

The trace guided verification is also a partial verification approach which can be seen as a hybrid solution between a debugging and a verification aid. It can be used to discover early bugs in the design cycle. *Spin* allows for instance to limit the search in the state space of the system to some fixed depth. But, if the system is erroneous in one of its phases after a (long) *initialization*, this limited search will not find the error.

We briefly present a technique which extends the principle of limited search; it forces the model checker to verify safety properties on certain areas in the state space that are believed to be *dangerous*. In order to do so, we suppose that an *interesting* trace can be obtained, either by automatic test case generation or by simulation or monitoring of the real system. Using this trace and a simple transformation of the system's specification, we make the model checker explore all states on that trace, and within a certain diameter of that trace (c.f. figure 7.1).

One advantage of this method is that we can target a precise part of the model. Therefore, the designer, who has extended knowledge of the system, can guide the model checker to the parts that are potentially dangerous. Another advantage of our method is that the error traces are generally short and easy to understand. Indeed, since we use an existing trace and the error is not too far away, the error trace can easily be mapped to the original input trace. We also show how *Spin* can be used as it is to perform this kind of exploration.

We first show how to transform PROMELA specifications to enable this technique. We then present a toy example of a faulty token ring leader election protocol [Lan77].

Guiding a PROMELA Process Using a Trace

A central concept in this technique is to determinize a non deterministic PROMELA specification. The aim is to make a PROMELA process go into the

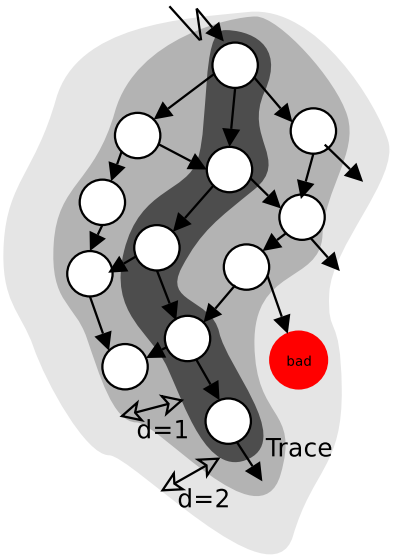

Figure 7.1: Trace centered state space exploration

direction of the trace. This is done by forcing its non deterministic actions to be the same (or within the diameter) of the ones dictated by the trace.

Supposing that non determinism is caused by interleaving, *if* and *do* statements, we will add a synchronization channel `sync` to each process that will dictate the choices the model has to make to respect the given trace. A non deterministic `if` choice :

```
if :: C1 -> A1;
   :: C2 -> A2;
fi
```

is determinized by the messages on this channel. If for example `A1` is the next action of the used trace, it can force the model to follow that path with the `sync!lbl_C1` in the transformed code, presented hereafter.

```
chan sync = [0] of { int }
if :: sync?lbl_C1 ->
      assert(C1); A1;
   :: sync_lbl_C2 ->
      assert(C2); A2;
fi
```

Note that traces have to be compatible with the model. On the previous example, incorrect traces could force the model to execute `A1` even if `C1` is not satisfied. The `assert(C1)` instruction checks for this possibility.

To enable the search up to a distance `MAXD` of the trace, a global variable `d` is added to the specification, and is incremented each time the specification makes a move away from the trace. If `d` reaches `MAXD`, the system is deadlocked and the model checker backtracks. Notice that forcing the model to follow the trace should only happen when `d` is equal to `0`. Figure 7.2 shows this transformation on an *if* and a *do* construct. The trace is represented by a PROMELA process performing a sequence of `d==0 -> synch!...` instructions guiding the model.

```
if :: C1 -> A1;                    do :: C1 -> A1;
   :: C2 -> A2;                       :: C2 -> A2;
fi                                 od
             ⇓                                  ⇓

                chan sync = [0] of { int };
                        int d = 0;
if :: d == 0 ->                    do :: d == 0 ->
      if :: sync?lbl_C1 ->               if :: sync?lbl_C1 ->
            assert(C1); A1;                    assert(C1); A1;
         :: sync?lbl_C2 ->                  :: sync?lbl_C2 ->
            assert(C2); A2;                    assert(C2); A2;
      fi;                                fi;
   :: d < MAXD -> d++;                :: d < MAXD -> d++;
      if :: C1 -> A1;                    if :: C1 -> A1;
         :: C2 -> A2;                       :: C2 -> A2;
      fi                                 fi
fi                                 od
```

Figure 7.2: Trace centered transformation of Promela specifications

Token Ring leader election

We present here a token ring leader election algorithm proposed in [Lan77]. Consider several stations that have to access a network in a mutual exclusive way. To keep access exclusive, a unique token travels between the stations, the owning station having the right to access the network. In this particular case, we take the model of 3 stations, communicating on lossy asynchronous channels with a bounded size of 3 messages. Since the channels may loose messages, the token can get lost and a new one has to be generated. Each process can be in a *eligible* and *non-eligible* state. The non-eligible state is the *normal* state of a process, where no election is needed. In this state, the

station can access the network when it has the token and when it is done, it passes the token to the next station. Claims are forwarded by a station in this state if they did not originate from that station. Note that a station in its non-eligible state can send a claim and then passes in eligible state. This models the timeout if a station did not receive a token for a certain period of time. The id of the station that started the election is added to this message. A station in eligible state receiving a claim behaves as follows :

1. If the id is smaller than its own, the station switches to eligible mode knowing that someone started the election.

2. If the id is greater than its own, the station destroys the claim, since the claim is definitely not going to win the election, and the station itself is in eligible state.

3. If the id is equal to its own, the station knows that all other stations transmitted its claim and that it is the new leader. It can therefore generate a new token and then changes its state to non-eligible.

An erroneous PROMELA specification of the election algorithm is given in figure 7.4, where a station generates a token even if it is not elected. (The error introduced in the specification is marked with `#ifdef ERROR_IN_MODEL`).

Note that with this example, it is hard to find a meaning for a *dangerous* trace. It is not our goal to promote our method on this example, but only use it to show how our technique behaves. We therefore consider random traces, generated from the example, by a random walk through the state space performed by *Spin* (cfr. figure 7.5). The output produced by the `printfs` during this random walk will be used to guide the exploration.

Figure 7.6 shows a snippet of the code transformed using the description given before. The previously generated trace, is put into a process and the complete system is verified.

The results of the tests are represented in figure 7.3, where L is the length of the trace, d is the diameter of the search, and the %-success is the mean number of times the model checker found the error (mean results for 50 instances). Remark that increasing length makes the search more accurate because of the probability that the trace is passing close to an error. Remind that using random traces is not the strength of this method : only *interesting* traces should be used during verification. Also note that, because more states are covered, increasing the diameter increases the chances of finding the error.

This first experiment shows that our approach can be useful, but, in its present form, it is far from being optimal. It has the advantage of being

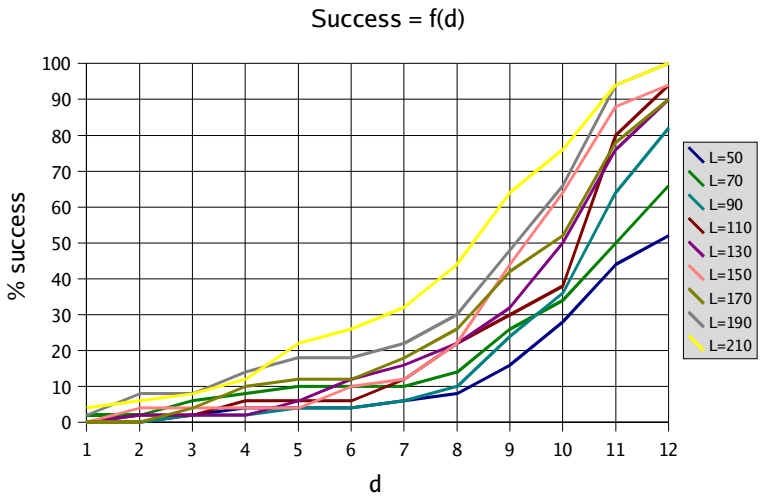

Figure 7.3: Results of the token ring example

applicable without the need to change the model checker. However, the drawbacks are serious. Introducing `d` in the model makes the method incomplete : even if the complete state space for the initial model could be searched in a reasonable amount of time and memory, the trace centered model will not. Indeed, remark that `d` is part of each state in the state space, which causes the state space to be unfolded with increasing values of `d`. In essence, if `d` may become arbitrary large, it introduction in the state descriptor makes finite systems become infinite. The same reason deteriorates the partial order reduction algorithm, since transitions are less likely to be independent. A direct implication of these two drawbacks causes greater memory usage than actually needed.

If `d` would be removed from the specification, and the model checker's search strategy changed to directly take into account `d`, these drawbacks would no longer be present. The complexity of such a search algorithm compared to standard depth first search is comparable, since the same number of states would be visited, and no state is visited twice. However, the set of *open* states, (i.e. states for which not all successors are visited) would be much larger compared to standard depth first search where the *open* states are only those that are on the search stack. In light of these remarks, an efficient implementation would therefore not take more time, but would be more memory consuming.

```
chan c_1_2 = [3] of { int, int };
show int network = 0;

inline station_eligible(c_in, c_out,
               my_id, state) {
  if :: // Receive a claim
        atomic {
        c_in ? CLAIM, id;
        if :: id < my_id ->
              c_out ! CLAIM, id;
              state = LEADER;
#ifdef ERROR_IN_PROTOCOL
           :: id >= my_id ->
#else
           :: id > myid -> skip;
           :: id == my_id ->
#endif
              state = LEADER;
              // New token
              c_out ! TOKEN, 0;
        fi
        }
     :: // Receive a token
        c_in  ? TOKEN, 0;
     :: // ReStart election
        c_out ! CLAIM, my_id;
  fi
}


inline station(c_in, c_out,
               my_id) {
 int id, a, b;
 byte state = LEADER;
 do
    :: state == ELECTION ->
       station_eligible(c_in, c_out,
            my_id, state);
    :: state == LEADER ->
       station_non_eligible(c_in, c_out,
            my_id, state);
    :: // Drop message
       c_in ? a, b;
  od
}
```

```
chan c_2_3 = [3] of { int, int };
chan c_3_1 = [3] of { int, int };

inline station_eligible(c_in,c_out,
              my_id, state) {
  if :: // Receive a claim
        atomic {
        c_in ? CLAIM, id ->
        if :: id != my_id ->
              c_out ! CLAIM, id;
           :: else;
        fi
        }
     :: // Receive a token
        atomic {
        c_in ? TOKEN, 0 ->
        network=network+1;
        }
        // Access to the network
        assert(network == 1);
        atomic {
        network=network-1;
        c_out ! TOKEN, 0;
        }
    :: // Start election
       c_out ! CLAIM, my_id;
       state = ELECTION;
  fi
}

active proctype p1() {
  station (c_3_1,c_1_2,1); }
active proctype p2() {
  station (c_1_2,c_2_3,2); }
active proctype p3() {
  station (c_2_3,c_3_1,3); }
```

Figure 7.4: Token Ring leader election in Promela

```
inline station_eligible(c_in, c_out, my_id, state) {
  if :: // Receive a claim
        atomic {
        c_in ? CLAIM, id;
        printf("d==0->sync%d!DO_RECEIVE_CLAIM;\n", my_id);
        if :: id < my_id ->
                                    ...
        fi
        }
     // Receive a token
     :: atomic {
            c_in ? TOKEN, 0;
            printf("d==0->sync%d!DO_RECEIVE_TOKEN;\n", my_id);
        }
     // Start election
     :: atomic {
            c_out ! CLAIM, my_id;
            printf("d==0->sync%d!DO_SEND_CLAIM;\n", my_id);
        }
  fi
}
```

Figure 7.5: Token Ring leader election producing a trace

```
chan sync = [0] of { int };

inline station_eligible(c_in, c_out, my_id, state) {
if   :: atomic{ d < MAXD -> d++; } // Explore diameter

        if :: // Receive a claim
              atomic {
              c_in ? CLAIM, id;
              if :: id < my_id ->
                                    ...
              fi
              }
           :: // Receive a token
              c_in  ? TOKEN, 0;
           :: // ReStart election
              c_out ! CLAIM, my_id;
    :: d==0 ->                      // Explore trace
       if :: // Receive a claim
             atomic {
             sync ? DO_RECEIVE_CLAIM;
             c_in ? CLAIM, id;
             if :: id < my_id ->
                                    ...
             fi
             }
          :: // Receive a token
             atomic {
                 sync  ? DO_RECEIVE_TOKEN;
                 c_in  ? TOKEN, 0;
             }
          :: // ReStart Election
             atomic {
                 sync  ? DO_SEND_CLAIM
                 c_out ! CLAIM, my_id;
             }
       fi
fi
```

Figure 7.6: Token Ring leader election transformed

Chapter 8

Conclusions

In this thesis, we studied the language and environment $_{\mathrm{d}}$SL. This language is designed upon an exiting industrial language used for the design of industrial controllers. $_{\mathrm{d}}$SL has two benefits with respect to the existing language : its automatic distribution allows the programmer to concentrate on the functional aspects of the controller's design, and its underlying formal semantics offers the possibility to formally verify the correctness of systems designed with $_{\mathrm{d}}$SL. Both aspects answered existing needs, and the implementation of $_{\mathrm{d}}$SL has therefore received a positive feedback from the industry.

The problems faced in this work are multiple :

1. designing $_{\mathrm{d}}$SL to allow transparent distribution, taking into account an already existing language
2. formalizing the semantics of $_{\mathrm{d}}$SL, studying the influence of the physical distribution on the behavior of a given $_{\mathrm{d}}$SL program
3. generating efficient executable code
4. implementing $_{\mathrm{d}}$SL's execution environment
5. exploiting $_{\mathrm{d}}$SL's formal semantics to formally verify the correctness of a $_{\mathrm{d}}$SL program

Each of these problems has been studied and solutions have been proposed in this work.

The design of $_{\mathrm{d}}$SL is based upon a study of solutions presented in the literature. An overview of the most important works is presented in chapter 2. We adopted a hybrid execution of atomic (non distributable) code which involves instantaneous reaction to events, together with sequential (distributable) code. For the execution of the sequential code we chose a

statically calculated thread migration which allows an easy implementation, insures stable performances and offers eased monitoring and debugging. Our approach of using a programming language used in the industry has a practical interest which should not be neglected. This language is presented in chapter 3, and we show how a few additions to this language enable the programmer to manage the co-existence of both sequential and atomic code.

The semantics of dSL is given as a labeled transition system, which is defined using structural operational semantics. We showed how the distribution influences the behavior of a given dSL program, more particularly, we related the physical distribution of captors and actuators in a dSL program to the behaviors of that program. For this purpose, we proved the existence of a lattice of distributions, and proved a corresponding lattice of behaviors. This strong result states that when a program is more distributed, its possible behaviors are extended with new behaviors, but include all those of the less distributed program. The existence of a maximal element in the lattice of distributions is therefore of capital importance, since its associated behavior contains the behaviors for any possible distribution.

The generation of the most efficient code is shown to be equivalent to solving NP-Complete problems. This bad news is tackled by introducing, in chapter 4, efficient heuristics approximating these problems. Two problems were addressed. First, we studied the production of efficient sequential distributed code, using a static estimation of the number of times each instruction is executed. Solving this problem is shown to be equivalent to solving the Multiterminal Cut Problem. The best combinatorial algorithmic solution previously known used a shrinkage technique, which allows to safely reduce the size of an instance of the Multiterminal Cut problem. We introduced the notion MAX-MIN-`st` cut, which allowed us on the one hand to generalize the existing shrinkage theorem, and on the other hand to design a fast heuristics which gives better results in practice. The second problem addressed consists in the reordering of instructions while maintaining an equivalent interface with the environment, in order to increase the performance during execution. This problem is related to the known Precedence Constrainted Class Sequencing Problem. We studied the efficiency of existing heuristics, and introduced a new heuristics which takes advantage of the structure of a given instance.

dSL's implementation, presented in chapter 5, is based on a framework using a compiler-distributer and a virtual machine. We covered in detail how the sequential code is handled using static flow analysis based on def-use chains.

Finally, in chapter 6, the formal verification of dSL programs is per-

formed using a translation from $_{\mathrm{d}}$SL to PROMELA, the specification language used by the model checker *Spin*. We showed how this translation can be done automatically, and illustrated our approach on three case studies. Although the state space explosion problem is a major problem for the verification of large $_{\mathrm{d}}$SL programs, we profit from the fact that the processing phase, which contains the most elaborate part of the controller's behavior, can be done in a single transition. This results in a major state space reduction which allow us to exhaustively verify moderate size programs.

We ended our study of $_{\mathrm{d}}$SL with a discussion on the missing features of $_{\mathrm{d}}$SL, and some future works. The missing features all are related to dynamic concepts, which are difficult to implement because of our design choice to use static distribution. Further work include the addition of these dynamic concepts,

Although several questions remain, $_{\mathrm{d}}$SL has reached a ready-to-use status, and fits the specific needs (i.e. transparent distribution, a way to formally verify $_{\mathrm{d}}$SL programs and industrial applicability) that where formulated during the first discussions on $_{\mathrm{d}}$SL.

Bibliography

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. ISBN 0-521-49619-5. Cambridge University Press, UK, 1996.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138(1):3–34, 1995.

[AD90] Rajeev Alur and D. L. Dill. Automata for modeling real-time systems. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 322–335, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[ADG+03] C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Rosu, and W. Visser. Experiments with test case generation and runtime analysis. In *Abstract State Machines, Advances in Theory and Practice, 10th International Workshop, ASM 2003*, volume 2589 of *LNCS*, pages 87–107, Tormina, Italy, March 2003. Springer.

[Aga76] V. N. Agafonov. On attribute grammars. In A. Mazurkiewicz, editor, *Mathematical Foundations of Computer Science*, volume 45 of *Lecture Notes in Computer Science*, pages 169–172, New York-Heidelberg-Berlin, September 1976. Springer-Verlag.

[AH94] Karl J. Astrom and Tore Hagglund. *Pid Controllers*. Number 1556175167 in ISBN. Instrumentation Systems, 1994.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Inc., Reading, Mass., 1986.

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the iso specification language lotos. *Comput. Netw. ISDN Syst.*, 14(1):25–59, 1987.

[Beg03] Laurent Van Begin. *Efficient Verification of Counting Abstractions for Parametric systems*. PhD thesis, Université Libre de Bruxelles, 2003.

[Ber98] G. Berry. The esterel v5 language primer. available at http://www.inria.fr/meije/esterel, March 1998.

[Ber99] G. Berry. The constructive semantics of pure esterel. Draft book, July 1999. available at http://www.inria.fr/meije/esterel/esterel-eng.html.

[Ber00] G. Berry. *The Foundations of Esterel*. MIT Press, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.

[BFR71] K.A. Baker, P.C. Fishburn, and F.S. Roberts. Partial orders of dimension 2. *Networks*, 2:11–28, 1971.

[BG92] Gerard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BL95] H. Brinksma and R. Langerak. Functionality decomposition by compositional correctness preserving transformation. *South African Computer Journal*, 13:2–13, 95.

[BLB93] Ed Brinksma, Rom Langerak, and Peter Broekroelofs. Functionality decomposition by compositional correctness preserving transformation. In *CAV*, pages 371–384, 1993.

[BLL+96] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 232–243, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[BMS97] F. Bonfattti, P.D. Monari, and U. Sampieri. *IEC 1131-3 programming methodology. Software engineering methods for industrial automated systems*. ISBN 2-9511585-0-5. CJ International Editions, 1997.

[Bri92] Preston Briggs. *Register allocation via graph coloring*. PhD thesis, Rice University, Houston, TX, USA, 1992.

[Buc60] J. R. Buchi. On a decision method in restricted second order arithmetic. In E. Nagel et al., editor, *Proceeding of the International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11, Stanford, CA, 1960.

[BZ83] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *ACM*, 30(2):323–342, 1983.

[CFSM] J.R. Correa, S. Fiorini, and N. Stier-Moses. A note on the precedence-constrained class sequencing problem. To be published.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–101, New York, NY, USA, 1982. ACM Press.

[CKR00] Gruia Calinescu, Howard Karloff, and Yuval Rabani. An improved approximation algorithm for multiway cut. *J. Comput. Syst. Sci.*, 60(3):564–574, 2000.

[CLR05] M. Costa, L. Letocart, and F. Roupin. Minimal multicut and maximal integer multiflow: a survey. *European Journal of Operational Research*, 162(1):55–69, 2005.

[CM94] John Joseph Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, September 1994.

[CMT99] I. Castellani, M. Mukund, and P. S. Thiagarajan. Synthesizing Distributed Transition Systems from Global Specification. In *Foundations of Software Technology and Theoretical Computer Science*, pages 219–231, 1999.

[CO96] Sunil Chopra and Jonathan H. Owen. Extended formulations for the a-cut problem. *Math. Program.*, 73:7–30, 1996.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. Conf Rec 14th Ann ACM Symp on Princ Prog Langs, 1987.

[Cun91] W.H. Cunningham. The optimal multiterminal cut problem. *DIMACS series in discrete mathematics and theoretical computer science*, 5:105–120, 1991.

[CVWY92] Constantin Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[Dar99] Alain Darte. On the complexity of loop fusion. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 149, Washington, DC, USA, 1999. IEEE Computer Society.

[DB95] R. Vohra D. Bertsimas, C. Teo. Nonlinear formulations and improved randomized approximation algorithms for multicut problems. In *Proc. 4th conference on integer programming and combinatorial optimization*, volume 920 of *LNCS*, pages 29–39, 1995.

[Dev04] Nicolas Devos. Génération de code de systemes distribués. Master's thesis, Université Libre de Bruxelles, 2004.

[DeWGM05] Bram De Wachter, Alexandre Genon, and Thierry Massart. From static code distribution to more shrinkage for the multiterminal cut. In Sotiris E. Nikoletseas, editor, *WEA*, volume 3503 of *LNCS*, pages 177–188. Springer, 2005.

[DeWGMM05] Bram De Wachter, Alexandre Genon, Thierry Massart, and Cédric Meuter. The formal design of distributed controllers with dsl and spin. *Formal Aspects of Computing*, 17(2):177–200, August 2005.

[DeWMM03a] Bram De Wachter, Thierry Massart, and Cédric Meuter. dsl: An environment with automatic code distribution for industrial control systems. In Marina Papatriantafilou and Philippe Hunel, editors, *OPODIS*, volume 3144 of *Lecture Notes in Computer Science*, pages 132–145. Springer, 2003.

[DeWMM03b] Bram De Wachter, Thierry Massart, and Cédric Meuter. An experiment on synthesis and verification of an industrial process control in the dsl environment. Proceedings of the 3rd Automated Verification of Critical Systems (AVoCS03), Technical Report DSSE-TR-2003-2, DSSE, Southampton (GB), April 2-3 2003.

[Die05] Reinhard Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer-Verlag, 2005.

[DJP+94] Elias Dahlhaus, David S. Johnson, Christos H. Papadimitriou, P. D. Seymour, and Mihalis Yannakakis. The complexity of multiterminal cuts. *SIAM J. Comput.*, 23(4):864–894, 1994.

[EC99] D.A. Peled E.M. Clarke, O. Grumberg. *Model Checking*. MIT Press, 1999.

[Edw00] Stephen A. Edwards. Compiling esterel into sequential code. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 322–327, New York, NY, USA, 2000. ACM Press.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, 1994.

[ELL01a] S. Edelkamp, A. Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI Symposium on Model-basedValidation of Intelligence*, 2001.

[ELL01b] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Directed explicit model checking with HSF–SPIN. *Lecture Notes in Computer Science*, 2057:57–79, 2001.

[ELL01c] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Trail-directed model checking. In Scott D. Stoller and Willem Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier, 2001.

[Eme90] E. Allen Emerson. *Temporal and modal logic*. MIT Press, Cambridge, MA, USA, 1990.

[Esk90] M. Rasit Eskicioglu. Design issues of process migration facilities in distributed systems. In *IEEE Computer Society Technical Committe on Operating Systems and Application Environments Newsletter*, volume 4, pages 3–13, 1990.

[FF62] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.

[FJJV97] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and Cesar Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29(1-2):123–146, 1997.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[GBJL02] Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. John Wiley, 2002.

[Gen04] Alexandre Genon. On the verification of dsl, a language to design distributed industrial control systems. Technical report, U.L.B., September 2004.

[Gir94] A. Girault. *Sur la Répartition de Programmes Synchrones*. Phd thesis, INPG, Grenoble, France, January 1994.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[God91] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 176–185, London, UK, 1991. Springer-Verlag.

[God05] Olivier Godart. Testing automatique de programmes de contrôle d'équipement industriels écrits en langage dsl. Master's thesis, Université Libre de Bruxelles, 2005.

[Gra66] R. L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, October 1988. ISSN:0004-5411.

[GV02] Alex Groce and Willem Visser. Model checking java programs using structural heuristics. In *International Symposium on Software Testing and Analysis*, pages 12–21, July 2002.

[GVY94] Naveen Garg, Vijay V. Vazirani, and Mihalis Yannakakis. Multiway cuts in directed and node weighted graphs. In *Proceedings of the 21st International Colloquium on Automata, Languages and Programming*, pages 487–498. Springer-Verlag, 1994.

[Haj57] G. Hajos. Uber eine art von graphen. *Intern. Math.*, 11, 1957.

[Ham50] Richard W. Hamming. Error-detecting and error-correcting codes. *Bell System Technical Journal*, 29(2):147–160, 1950.

[HHG99] J. Hennessy, M. Heinrich, and A. Gupta. Cache-coherent distributed shared memory: Perspectives on its development and future challenges. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):418–429, 1999.

[HHLV97] Lisa Hollermann, Tsan-sheng Hsu, Dian Rae Lopez, and Keith Vertanen. Scheduling problems in a practical allocation model. *J. Comb. Optim.*, 1(2):129–149, 1997.

[HHW97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):110–122, 1997.

[HK01] K. Hogstedt and D. Kimelman. Graph cutting algorithms for distributed applications partitioning. *SIGMETRICS Performance Evaluation Review*, 28(4):27–29, 2001.

[HLLR00] Tsan-sheng Hsu, Joseph C. Lee, Dian Rae Lopez, and William A. Royce. Task allocation on a network of processors. *IEEE Trans. Computers*, 49(12):1339–1353, 2000.

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. IEEE Transactions Syst. Science and Cybernetics 4(2):100-107, 1968.

[Hol03] G. J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003. ISBN: 0321228626.

[HP00] Michael Hind and Anthony Pioli. Which pointer analysis should i use? In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 113–123, New York, NY, USA, 2000. ACM Press.

[IH97] Marc Shapiro II and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[JC02] H. Jiang and V. Chaudhary. On improving thread migration: Safety and performance. In *Proceedings: 9th International Conference on High Performance Computing 2002*, volume 2552 of *LNCS*, pages 474–484, Berlin, Germany, December 2002. Springer-Verlag.

[JCTG96] J. C. Fernandez, C. Jard, T. Jéron, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 348–359, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[Jér91] Thierry Jéron. Testing for unboundedness of FIFO channels. In *STACS 91: Proceedings of the 8th annual symposium on Theoretical aspects of computer science*, pages 322–333, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[JJ93] Thierry Jéron and Claude Jard. Testing for unboundedness of FIFO channels. *Theor. Comput. Sci.*, 113(1):93–117, 1993.

[KKS+99] David R. Karger, Philip Klein, Cliff Stein, Mikkel Thorup, and Neal E. Young. Rounding algorithms for a geometric embedding of minimum multiway cut. In *STOC '99: Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 668–678, 1999.

[KM94] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and

distribution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320, London, UK, 1994. Springer-Verlag.

[KMC88] J. Misra K. Mani Chandy. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., 1988.

[KU96] Anton P. Karadimce and Susan Darling Urban. Refined triggering graphs: A logic-based approach to termination analysis in an active object-oriented database. In *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*, pages 384–391, Washington, DC, USA, 1996. IEEE Computer Society.

[Lam79] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers, 28(9):690–691*, sept. 1979.

[Lan77] G. Le Lann. Towards a formal approach. *Distributed Systems, IFIP Congress*, pages 155–160, 1977.

[Lan90] Rom Langerak. Decomposition of functionality: a correctness-preserving lotos transformation. In *Proceedings of the IFIP WG6.1 Tenth International Symposium on Protocol Specification, Testing and Verification X*, pages 229–242. North-Holland, 1990.

[LCL88] F. J. Lin, P. M. Chu, and M. T. Liu. Protocol verification using reachability analysis: the state space explosion problem and relief strategies. In *Proceedings of the ACM workshop on Frontiers in computer communications technology*, pages 126–135. ACM Press, 1988.

[LGLL91] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real time applications with signal. Proceedings of the IEEE, 79(9):1321-1336, September 1991.

[LKB77] Lenstra, J. and A. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[LMT91] Christopher B. Lofgren, Leon F. McGinnis, and Craig A. Tovey. Routing printed circuit cards through an assembly cell. *Oper. Res.*, 39(6):992–1004, 1991.

[LMW04] S. Leue, R. Mayr, and W. Wei. A scalable incomplete test for the boundedness of UML RT models. In K. Jensen and A. Pdelski, editors, *Proceedings: 10th International Conference, TACAS 2004*, volume 2988 of *LNCS*, pages 327–341, Barcelona, Spain, March 2004. ETAPS 2004, Springer-Verlag.

[Lof86] C.B. Lofgren. *Machine configuration of flexible printed circuit board assembly systems.* PhD thesis, School of ISyE, Georgia Institute of Technology, Atlanta (GA), USA, 1986.

[LW66] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[Mas92] T. Massart. A calculus to define correct transformations of LOTOS specifications. In *Proceedings of the FORTE'91 conference*, pages 281–296, 1992.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In Cosnard M. et al., editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).

[McM93] K. McMillan. *Symbolic Model Checking.* Kluwer Academic Press, 1993.

[Mic05] Nicolas Micheli. Design and verification of real-time distributed industrial control systems. Master's thesis, Université Libre de Bruxelles, 2005.

[Mil81] R. Milner. On relating synchrony and asynchrony. Technical Report CSR-75-80, Computer Science Dept., Edinburgh Univ., 1981.

[Mil89] R. Milner. *Communication and Concurrency.* PHI Series in Computer Science. Prentice Hall, 1989.

[ML87] Michael Marcotty and Henry Ledgard. *The world of programming languages*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[Mor99] René Morin. Decompositions of Asynchronous Systems. In *Proc. CONCUR'98, Springer Lect. Notes in Comp. Sci. 1466*, pages 549–565. Springer, 1999.

[Mos93] David Mosberger. Memory consistency models. *Operating Systems Review*, 27(1):18–26, 1993.

[MP97] C. Morin and I. Puaut. A survey of recoverable distributed shared memory systems. *IEEE Trans. on Parallel and Distributed Systems*, 8(9):959–969, 1997.

[NC00] Gleb Naumovich and Lori A. Clarke. Classifying properties: an alternative to the safety-liveness classification. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 159–168, New York, NY, USA, 2000. ACM Press.

[NL91] B. Nizeberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. IEEE Computer, vol. 24, no.8, pp. 52-60, Aug. 1991.

[Nta88] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng.*, 14(6):868–874, 1988.

[NZ01] J. Naor and L. Zosin. A 2-approximation algorithm for the directed multiway cut problem. *SIAM J. Comput.*, 31(2):477–482, 2001.

[Pat05] Ron Patton. *Software Testing*. Sams, 2005. ISBN : 0672327988.

[Pel94] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer-Verlag.

[PW72] Leon Presser and John R. White. Linkers and loaders. *ACM Comput. Surv.*, 4(3):149–167, 1972.

[RCT92] RCTA/DO-178B. *Software Considerations in Airborne Systems and Equipment Certification*, December 1992.

[RU81] K.-J. Raiha and E. Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *J. THEOR-COMP-SCI*, 16(2):187–198, November 1981.

[SC91] M.R. Rao S. Chopra. On the multiway cut polyhedron. *Networks*, 21:51–89, 1991.

[SEM03] Alin Stefănescu, Javier Esparza, and Anca Muscholl. Synthesis of distributed algorithms using asynchronous automata. In *CONCUR*, 2003.

[Tov04] Craig A. Tovey. Non-approximability of precedence-constrained sequencing to minimize setups. *Discrete Appl. Math.*, 134(1-3):351–360, 2004.

[Val91] Antti Valmari. A stubborn attack on state explosion. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 156–165, London, UK, 1991. Springer-Verlag.

[VB02] Sergiy A. Vilkomir and Jonathan P. Bowen. From MC/DC to RC/DC: Formalization and analysis of control-flow testing criteria. Technical Report SBU-CISM-02-17, South Bank University, CISM, London, UK, 2002.

[WWWK94] J. Wald, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc., M/S 29-01, 2550 Garcia Avenue, Mountainview, CA 94043, November 1994.

[YD98] C. Han Yang and David L. Dill. Validation with guided search of the state space. In *Design Automation Conference*, pages 599–604, 1998.

Appendix A

SL syntax

program	::=	uses_declaration data_type_declaration global_program_var_declarations global_statement_list "**PROGRAM**" ID program_var_declarations statement_list "**END_PROGRAM**"
uses_declaration	::=	["**USES**" uses_list "**END_USES**"]
uses_list	::=	[ID "**;**"]
data_type_declaration	::=	["**TYPE**" type_declaration_list "**END_TYPE**"]
type_declaration_list	::=	{ type_declaration "**;**" }
rule_statement	::=	when_statement
	|	global_globalisation_statement
	|	typed_when_statement
	|	typed_globalisation_statement
when_statement	::=	"**WHEN**" expression "**THEN**" local_variables statement_list "**END_WHEN**"
typed_when_statement	::=	"**WHEN**" "**IN**" simple_specification expression "**THEN**" local_variables statement_list "**END_WHEN**"
global_globalisation_statement	::=	"**CONTROL**" variable "**WITH**" expression
typed_globalisation_statement	::=	"**CONTROL**" "**IN**" simple_specification variable "**WITH**" expression
statement_list	::=	{ statement "**;**" } statement "**;**"

```
statement                   ::= [ "NIL" | assignment_statement |
                                method_call_statement | call_all_statement
                                | iteration_statement | selection_statement
                                | print_statement | error_statement |
                                say_statement | log_statement |
                                alias_statement | nature_statement |
                                domain_statement |
                                set_push_variable_statement ]
print_statement             ::= "PRINT" value_list
error_statement             ::= "ERROR" value_list
say_statement               ::= "SAY" value_list
log_statement               ::= "LOG" value_list
iteration_statement         ::= while_statement
                              | repeat_statement
                              | for_statement
                              | exit_statement
selection_statement         ::= if_statement
                              | case_statement
method_call_statement       ::= variable "<-" ID "(" value_list ")"
assignment_statement        ::= variable ":=" expression
call_all_statement          ::= "ALL" simple_specification "<-" ID "("
                                value_list ")"
alias_statement             ::= "ALIAS" expression expression
                                expression
nature_statement            ::= "SET_NATURE" variable expression
domain_statement            ::= "SET_DOMAIN" variable expression
set_push_variable_statement ::= "SET_PUSH_VARIABLE" variable
                                expression expression expression
do_particle                 ::= "DO"
until_particle              ::= "UNTIL"
exit_statement              ::= "EXIT"
while_statement             ::= "WHILE" expression do_particle
                                statement_list "END_WHILE"
repeat_statement            ::= "REPEAT" statement_list until_particle
                                expression "END_REPEAT"
for_statement               ::= "FOR" variable ":=" expression "TO"
                                expression ( do_particle statement_list
                                "END_FOR" | "BY" expression
                                do_particle statement_list "END_FOR" )
elsif_block                 ::= "ELSIF" expression "THEN"
                                statement_list ( "ENDIF" | "ELSE"
                                statement_list "ENDIF" | elsif_block )
if_statement                ::= "IF" expression "THEN" statement_list
                                ( "ENDIF" | "ELSE" statement_list
                                "ENDIF" | elsif_block )
signed_integer_constant     ::= UNSIGNED_INTEGER
```

	|	INTEGER
	|	"−" UNSIGNED_INTEGER
	|	"+" UNSIGNED_INTEGER
case_list_element	::=	signed_integer_constant [":" signed_integer_constant]
case_list	::=	{ case_list_element "," } case_list_element
case_element	::=	case_list ":" statement_list
list_of_case_elements	::=	{ case_element } case_element
case_statement	::=	"**CASE**" expression "**OF**" list_of_case_elements ("**END_CASE**" | "**ELSE**" statement_list "**END_CASE**")
value_list	::=	[expression ["," value_list]]
int_rval	::=	add_expression
expression	::=	or_expression
equal_operator	::=	"="
	|	"<>"
comparison_operator	::=	"<"
	|	">"
	|	"<="
	|	">="
comparison	::=	equ_expression { equal_operator equ_expression }
equ_expression	::=	add_expression { comparison_operator add_expression }
or_operator	::=	"**OR**"
or_expression	::=	xor_expression { or_operator xor_expression }
xor_operator	::=	"**XOR**"
xor_expression	::=	and_expression { xor_operator and_expression }
and_operator	::=	"**AND**"
	|	"&"
and_expression	::=	comparison { and_operator comparison }
add_expression	::=	{ term ("+" | "−") } term
term	::=	power_expression
	|	term "∗" unary_expression
	|	term "/" unary_expression
	|	term "**MOD**" unary_expression
power_expression	::=	unary_expression ["**POWER**" unary_expression]
unary_expression	::=	primary_expression
	|	"−" primary_expression
	|	"+" unary_expression
	|	"**NOT**" unary_expression
	|	"**IS_UNKNOWN**" unary_expression
	|	"**ADDRESS**" unary_expression

primary_expression	::=	INTEGER
	|	UNSIGNED_INTEGER
	|	LONG_INTEGER
	|	UNSIGNED_LONG_INTEGER
	|	FLOATING_POINT
	|	"**TRUE**"
	|	"**FALSE**"
	|	"**UNKNOWN_BOOL**"
	|	"**UNKNOWN_SINT**"
	|	"**UNKNOWN_INT**"
	|	"**UNKNOWN_DINT**"
	|	"**UNKNOWN_LINT**"
	|	"**UNKNOWN_USINT**"
	|	"**UNKNOWN_UINT**"
	|	"**UNKNOWN_UDINT**"
	|	"**UNKNOWN_ULINT**"
	|	"**UNKNOWN_REAL**"
	|	"**UNKNOWN_LREAL**"
	|	"**UNKNOWN_STRING**"
	|	"**UNKNOWN_TIME**"
	|	"**UNKNOWN_TIME_OF_DAY**"
	|	"**UNKNOWN_DATE_AND_TIME**"
	|	"**UNKNOWN_DATE**"
	|	"**NULL**"
	|	QUOTED_STRING
	|	TIME_CONSTANT
	|	DATE_CONSTANT
	|	DATE_AND_TIME_CONSTANT
	|	TIME_OF_DAY_CONSTANT
	|	variable
	|	"(" expression ")"
	|	function_call
function_call	::=	function_name "(" value_list ")"
function_name	::=	ID
variable	::=	variable "^"
	|	variable "." ID
	|	variable "[" int_rval "]"
	|	ID
global_variable_declarations	::=	"**GLOBAL**" a_var_decl_list "**END_VAR**"
other_var_declarations	::=	var_declarations
program_var_declaration	::=	global_variable_declarations
	|	other_var_declarations
program_var_declarations	::=	{ program_var_declaration }
global_program_var_declarations	::=	{ global_variable_declarations }

```
a_var_decl_list                 ::=  a_var_decl { ";" a_var_decl }
a_var_decl                      ::=  [ ID ( ":" type_specification | ","
                                     a_var_decl ) ]
structure_specification         ::=  structure_declaration
enum_specification              ::=  enum_declaration
enum_declaration                ::=  "(" enum_element_declaration_list ")"
enum_element_name               ::=  ID
enum_element_declaration        ::=  enum_element_name
enum_element_declaration_list   ::=  enum_element_declaration { ","
                                     enum_element_declaration }
sub_range                       ::=  signed_integer_constant ":"
                                     signed_integer_constant
sub_range_list                  ::=  sub_range { "," sub_range }
type_specification              ::=  array_specification
                                   | structure_specification
                                   | pointer_specification
                                   | list_specification
                                   | enum_specification
                                   | simple_specification
array_specification             ::=  "ARRAY" "[" sub_range_list "]" "OF"
                                     type_specification
list_specification              ::=  "LIST" "[" UNSIGNED_INTEGER "]"
                                     "OF" type_specification
pointer_specification           ::=  "POINTER" "TO" type_specification
structure_declaration           ::=  "STRUCT"
                                     structure_element_declaration_list
                                     "END_STRUCT"
structure_element_name          ::=  ID
structure_element_declaration   ::=  structure_element_name ":"
                                     type_specification
structure_element_declaration_list ::=  structure_element_declaration ";" [
                                     structure_element_declaration_list ]
a_type_name                     ::=  ID
type_declaration                ::=  a_type_name ":" type_specification
global_statement                ::=  method_declaration
                                   | rule_statement
global_statement_list           ::=  { global_statement }
local_variables                 ::=  [ "VAR" a_var_decl_list "END_VAR" ]
var_declarations                ::=  "VAR" a_var_decl_list "END_VAR"
method_declaration              ::=  "METHOD" simple_specification "::"
                                     ID "(" a_var_decl_list ")" ( local_variables
                                     statement_list "END_METHOD" |
                                     "FORWARD" )
simple_specification            ::=  "BOOL"
                                   | "SINT"
                                   | "INT"
```

| “**DINT**”
| “**LINT**”
| “**USINT**”
| “**UINT**”
| “**UDINT**”
| “**ULINT**”
| “**REAL**”
| “**LREAL**”
| “**TIME**”
| “**TIME_OF_DAY**”
| “**DATE_AND_TIME**”
| “**DATE**”
| “**STRING**”
| ID

Remarks

[T] Means 1 at least one T.
{ T } Means T or ϵ
“**T**” is a literal.
T is a terminal.

Appendix B

dSL syntax

root	::=	dsl_program
	|	localizable_lhside_list
dsl_program	::=	declaration_list
declaration_list	::=	{ declaration }
declaration	::=	class
	|	global_variables
	|	site
	|	when
	|	when_in
	|	method
	|	sequence
class	::=	"**CLASS**" ID variable_declaration_list "**END_CLASS**"
global_variables	::=	"**GLOBAL_VAR**" variable_declaration_list "**END_VAR**"
local_variables	::=	["**LOCAL_VAR**" variable_declaration_list "**END_VAR**"]
variable_declaration_list	::=	{ variable_declaration "**;**" }
variable_declaration	::=	non_empty_id_list "**:**" type
non_empty_id_list	::=	ID { "**,**" ID }
site	::=	"**SITE**" ID localization_list "**END_SITE**"
localization_list	::=	{ localization "**;**" }
localization	::=	kind localizable_lhside "**:**" NUMBER "**.**" NUMBER "**.**" NUMBER
localizable_lhside	::=	ID
	|	localizable_lhside "**.**" ID
	|	localizable_lhside "[" constant "]"
localizable_lhside_list	::=	{ localizable_lhside }
kind	::=	"**INPUT**"
	|	"**OUTPUT**"

when ::= "**WHEN**" rhside "**THEN**"
local_variables w_instruction_list
"**END_WHEN**"

when_in ::= "**WHEN**" "**IN**" ID rhside "**THEN**"
local_variables w_instruction_list
"**END_WHEN**"

method ::= "**METHOD**" ID "**::**" ID "**(**"
parameter_declaration_list "**)**"
local_variables instruction_list
"**END_METHOD**"

sequence ::= "**SEQUENCE**" ID ("**(**"
parameter_declaration_list "**)**"
local_variables s_instruction_list
"**END_SEQUENCE**" | "**::**" ID "**(**"
parameter_declaration_list "**)**"
local_variables s_instruction_list
"**END_SEQUENCE**")

parameter_declaration_list ::= [variable_declaration ["**,**"
parameter_declaration_list]]

type ::= "**LONG**"
| "**INT**"
| "**BOOL**"
| ID
| "**ARRAY**" "[" NUMBER "**:**" NUMBER
"]" "**OF**" type

instruction_list ::= { instruction "**;**" }

w_instruction_list ::= { w_instruction "**;**" }

s_instruction_list ::= { s_instruction "**;**" }

instruction ::= while
| w_instruction

s_instruction ::= instruction
| wait

w_instruction ::= assign
| if
| call

assign ::= lhside "**:=**" rhside

wait ::= "**WAIT**" rhside

if ::= "**IF**" rhside "**THEN**" instruction_list else

else ::= ["**ELSE**" instruction_list] "**END_IF**"

while ::= "**WHILE**" rhside "**DO**" instruction_list
"**END_WHILE**"

lhside ::= ID
| lhside "**.**" ID
| lhside "[" rhside "]"

rhside ::= lhside
| constant

	|	"(" rhside ")"
	|	rhside "**OR**" rhside
	|	rhside "**AND**" rhside
	|	rhside "<" rhside
	|	rhside ">" rhside
	|	rhside "<=" rhside
	|	rhside ">=" rhside
	|	rhside "<>" rhside
	|	rhside "=" rhside
	|	rhside "**MOD**" rhside
	|	rhside "+" rhside
	|	rhside "−" rhside
	|	rhside "∗" rhside
	|	rhside "/" rhside
	|	"**NOT**" rhside
	|	"−" rhside
	|	"**IS_UNKNOWN**" rhside
	|	"∼" lhside
constant	::=	"**TRUE**"
	|	"**FALSE**"
	|	NUMBER
call	::=	(lhside "<−" | "**LAUNCH**" lhside "<−" | "**LAUNCH**") ID "(" rhside_list ")"
rhside_list	::=	[rhside ["**,**" rhside_list]]

Remarks

[T] Means 1 at least one T.
{ T } Means T or ϵ
"**T**" is a literal.
T is a terminal.

Appendix C

Canal locks controller source code

```
CLASS Gate
  motor_direction, motor_command, opened, closed,
  order_given : BOOL;
  button_open, button_close : BOOL;
END_CLASS

CLASS Lock
  water_up, water_down, water_command, water_direction,
  water_order_given : BOOL;
  bottom_gate, top_gate : GATE;
  button_fill, button_empty : BOOL;
END_CLASS

GLOBAL_VAR
  lock1, lock2                          : Lock;
  not_allowed_led                       : BOOL;
END_VAR

(* Gates *)
METHOD GATE::move(direction : BOOL)
  self.motor_direction := direction;
  self.motor_command := TRUE;
END_METHOD

METHOD GATE::reset_order_given()
  self.order_given := FALSE;
  not_allowed_led := FALSE;
END_METHOD

(* Equivalent to WHEN G.closed OR G.opened THEN ...
   for every object G of class GATE *)
WHEN IN GATE self.closed OR self.opened THEN (* W1 = lock1.bottom_gate *)
  self.motor_command := FALSE;               (* W2 = lock1.top_gate    *)
  LAUNCH self<-reset_order_given();          (* W3 = lock2.bottom_gate *)
END_WHEN                                     (* W4 = lock2.top_gate    *)
```

```
(* Locks *)
METHOD LOCK::water_move(direction : BOOL)
  IF NOT self.water_down THEN
    self.water_command := TRUE;
    self.water_direction := direction;
  END_IF;
END_METHOD

METHOD LOCK::reset_water_order_given()
  self.water_order_given := FALSE;
  not_allowed_led := FALSE;
END_METHOD

WHEN IN LOCK self.water_up OR self.water_down THEN   (* W5 = self -> lock1 *)
  self.water_command := FALSE;                       (* W6 = self -> lock2 *)
  LAUNCH self<-reset_water_order_given();
END_WHEN



WHEN lock1.bottom_gate.button_open THEN              (* W7 *)
  IF (~lock1.top_gate.closed) AND (NOT lock1.top_gate.order_given) AND
     (~lock1.water_down) AND (NOT lick1.water_order_given)
  THEN
    not_allowed_led := FALSE;
    lock1.bottom_gate.order_given := TRUE;
    LAUNCH lock1.bottom_gate<-move(TRUE);  (*open*)
  ELSE
    not_allowed_led := TRUE;
  END_IF;
END_WHEN

WHEN lock1.top_gate.button_open THEN                 (* W8 *)
  IF (~lock1.bottom_gate.closed) AND (NOT lock1.bottom_gate.order_given) AND
     (~lock2.bottom_gate.closed) AND (NOT lock2.bottom_gate.order_given) AND
     (~lock1.water_up) AND (NOT lock1.water_order_given)
  THEN
    not_allowed_led := FALSE;
    lock1.top_gate.order_given := TRUE;
    LAUNCH lock1.top_gate<-move(TRUE);   (*open*)
  ELSE
    not_allowed_led := TRUE;
  END_IF;
END_WHEN

WHEN lock1.bottom_gate.button_close THEN             (* W9 *)
  LAUNCH lock1.bottom_gate<-move(FALSE);  (*close*)
END_WHEN

WHEN lock1.top_gate.button_close THEN                (* W10 *)
  LAUNCH lock1.top_gate<-move(FALSE);     (*close*)
END_WHEN

WHEN lock1.button_fill THEN                          (* W11 *)
  IF (~lock1.bottom_gate.closed) AND (NOT lock1.bottom_gate.order_given) AND
     (~lock1.top_gate.closed) AND (NOT lock1.top_gate.order_given)
```

```
  THEN
    not_allowed_led := FALSE;
    lock1.water_order_given := TRUE;
    LAUNCH lock1<-water_move(TRUE);
  ELSE
    not_allowed_led := TRUE;
  END_IF;
END_WHEN

WHEN lock1.button_empty THEN                          (* W12 *)
  IF (~lock1.bottom_gate.closed) AND (NOT lock1.bottom_gate.order_given) AND
     (~lock1.top_gate.closed) AND (NOT lock1.top_gate.order_given)
  THEN
    not_allowed_led := FALSE;
    lock1.water_order_given := TRUE;
    LAUNCH lock1<-water_move(FALSE);
  ELSE
    not_allowed_led := TRUE;
  END_IF;
END_WHEN

WHEN lock2.bottom_gate.button_open THEN               (* W13 *)
  ... (* Same as W8, replace lock1 with lock2 ;
         switch top, bottom; switch up, down *)

WHEN lock2.top_gate.button_open THEN                  (* W14 *)
  ... (* Same as W7, replace lock1 with lock2 ;
         switch top, bottom; switch up, down *)

WHEN lock2.bottom_gate.button_close THEN              (* W15 *)
  LAUNCH lock2.bottom_gate<-move(FALSE);  (*close*)
END_WHEN

WHEN lock2.top_gate.button_close THEN                 (* W16 *)
  LAUNCH lock2.top_gate<-move(FALSE);     (*close*)
END_WHEN

WHEN lock2.button_fill THEN                           (* W17 *)
  ... (* Same as W11, replace lock1 with lock2 *)

WHEN lock2.button_empty THEN                          (* W18 *)
  ... (* Same as W12, replace lock1 with lock2 *)

(* main program *)
SEQUENCE init
  not_allowed_led := FALSE;
END_SEQUENCE
```

Appendix D

The dSL compiler-distributer frontend

In this appendix we discuss the implementation details of the frontend part of the compiler-distributer. The frontend is depicted in the global workflow of the compiler distributer which is represented in figure D.1.

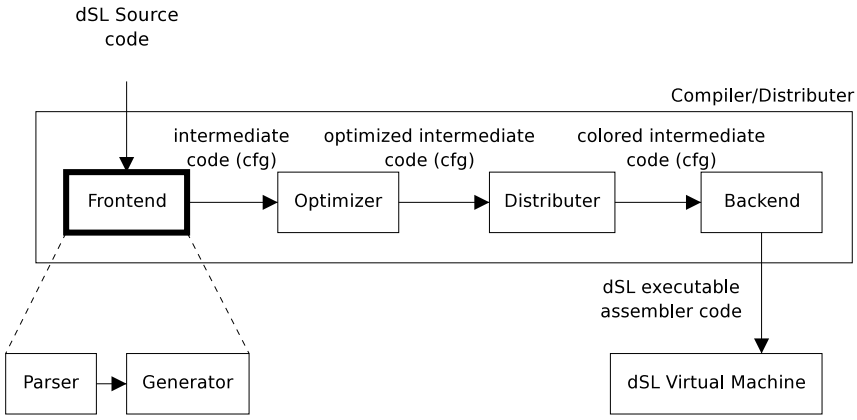

Figure D.1: The frontend in the compiler-distributer workflow

The parser is written in C++ using Lex and Yacc. It first constructs a global symbol table, in which all program elements are stored. During this process, syntactic and semantic checks are performed. Next, all `WHEN IN` declarations are translated into `WHEN`s. Finally, the parser generates intermediate code. The generator outputs a text file which is of the following form :

intermediate ::= "**BEGIN_DSL_INTERMEDIATE_CODE**" classlist arraylist gvarlist sitelist methodlist sequencelist whenlist "**END_DSL_INTERMEDIATE_CODE**"

classlist ::= "**BEGIN_DSL_CLASSES** [" *nbr_classes* "]" (class)* "**END_DSL_CLASSES**"
class ::= "**CLASS**" id name "[" *nbr_fields* "]" fieldlist
fieldlist ::= (id name type_id "**LINE**" *line_nbr*)*
arraylist ::= "**BEGIN_DSL_ARRAYS** [" *nbr_arrays* "]" (array)* "**END_DSL_ARRAYS**"
array ::= "**ARRAY** [" *from* "**:**" *to* "] **OF**" *type_id* "**LINE**" *line_nbr*

gvarlist ::= "**BEGIN_DSL_GLOBAL_VARIABLES** [" *nbr_gvars* "]" (var_decl)* "**END_DSL_GLOBAL_VARIABLES**"
var_decl ::= *id name type_id* "**LINE**" *line_nbr*

sitelist ::= "**BEGIN_DSL_SITES** [" *nbr_sites* "]" (site)* "**END_DSL_SITES**"
site ::= "**LOCALIZE**" lhside *card rack slot* "**LINE**" *line_nbr*

methodlist ::= "**BEGIN_DSL_METHODS** [" *nbr_methods* "]" (method)* "**END_DSL_METHODS**"
method ::= "**METHOD**" *id name* "[" *nbr_params* "/" *nbr_local_vars* "] **LINE**" *line_nbr* paramlist lvarlist cfg

paramlist ::= "**PARAM**" (var_decl)*

lvarlist ::= "**LOCAL_VAR**" (var_decl)*

cfg ::= "**BEGIN_CFG**" (node)* "**END_CFG**"
node ::= "**NEW_NODE**" *id* (instr)*

instr ::= (assign | if | jmp | | call | wait) "**LINE**" *line_nbr*
assign ::= "**ASSIGN**" lhside rhside
if ::= "**IF**" rhside *then_node else_node end_if_node* "**MEAN"** *proba_then proba_else proba_end_if*
jmp ::= "**JMP**" *node* "**MEAN"** *proba*
call ::= "**CALL**" *id* ("**SYNCH**" | "**ASYNCH**") "[" *nbr_params* "] **PARAM**" (rhside)* "**END_CALL**"
wait ::= "**WAIT**" rhside

rhside	::=	lhside | *constant* | unop rhside | binop rhside rhside
lhside	::=	field_sel | array_sel | ("**G**" | "**L**") *id*
field_sel	::=	"**.**" *constant* rhside
array_sel	::=	"[]" rhside rhside
unop	::=	"**-**" | "**IS_UNKNOWN**" | "∼" | "**NOT**"
binop		"**OR**" | "**AND**" | "<" ">" | "<=" | ">=" "**EQ**" | "**NEQ**" | "+" "−" | "∗" | "/" | "**MOD**"
sequencelist	::=	"**BEGIN_DSL_SEQUENCES** [" *nbr_seqs* "]" (sequence)* "**END_DSL_SEQUENCES**"
sequence	::=	"**SEQUENCE**" *id name* "[" *nbr_params* "/" *nbr_local_vars* "] **LINE**" *line_nbr* paramlist lvarlist cfg
whenlist	::=	"**BEGIN_DSL_WHENS** [" *nbr_whens* "]" (when)* "**END_DSL_WHENS**"
sequence	::=	"**WHEN**" *id name* "[" *nbr_local_vars* "] **LINE**" *line_nbr* "**CONDITION**" rhside lvarlist cfg

Remark that the parser already transforms the code inside `SEQUENCE`s and `METHODS` into control flow graphs (called *cfg*) which contain basic blocks (called *nodes* in the intermediate representation). Possible instructions produced by the parser are : assignment, conditional jump (called *if* in the intermediate representation), unconditional jump, call, and wait. Additionally, the parser simplifies the work by transforming complex expressions into prefix (also called *polish*) notation, and by using numerical identifiers instead of symbolic names. The distributer, who is next in the workflow, can read this intermediate code much more easily than the original dSL source code, and does not have to worry about syntactical correctness.

Also note that it is possible to give an estimation of the weighted control flow introduced in chapter 4. These values are indicated with the keyword `MEAN` in the intermediate representation. These values, however, are not really estimated by the current implementation of the parser. Instead, the parser estimates the probability of the `THEN` and `ELSE` branches of an `IF` instruction to `49.5%`, while the probability of `UNKNOWN` is estimated to `1 %`. The number of times a `WHILE` is executed is estimated to 100 by the parser. Obviously these values (and especially the k value for the `WHILE` instruction) should be estimated more correctly. Remember that these values are multiplied in the case of nested `IF` and `WHILE` instructions.

Appendix E

The dSL compiler-distributer distributer

In this appendix we discuss the implementation details of the distributer part of the compiler-distributer. The distributer is depicted in the global workflow of the compiler distributer which is represented in figure E.1.

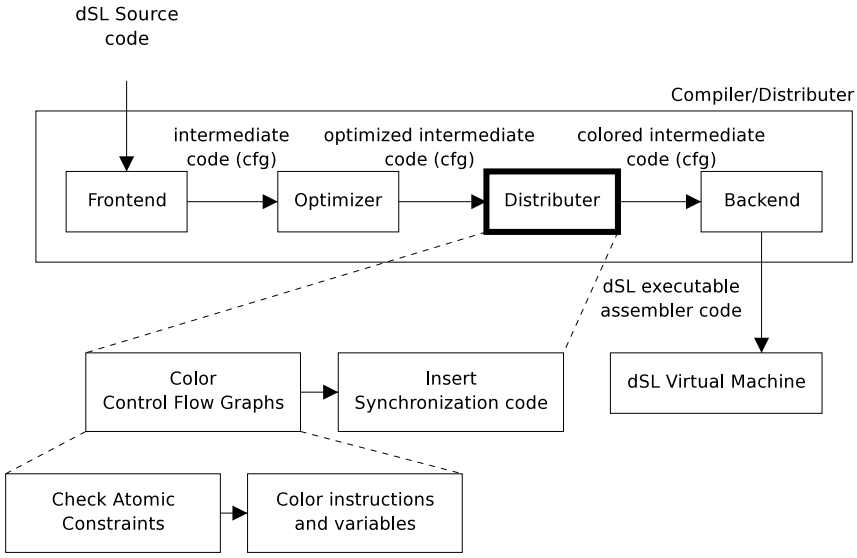

Figure E.1: The distributer in the compiler-distributer workflow

The coloring algorithm

The coloring algorithm used by the distributer is the one presented in chapter 4. We give some technical details here.

The first step in coloring all instructions and variables consists in constructing the sequential color graph presented in section 4.3.2. In the implementation, this graph is called dependency graph, for historical reasons. Recall that the atomicity constraints are expressed in this graph by edges of infinite weight. The graph is constructed using the instructions and variables from each control flow graph produced by the parser. There is a pointer in each variable in the compiler's symboltable and each instruction in the control flow graphs to the a node in the dependency graph, and vice versa.

When a control flow graph is encountered which is marked *atomic*, edges of infinite weight are inserted between all instructions and variables involved. Next, synchronous calls are inspected and more infinite edges are inserted, conform to the definition of synchronous flow (definition 20). Once the dependency graph is entirely constructed, traversal using only infinite weighted edges of the dependency graph is performed. If two nodes are found with different color in the same connected component, an error is issued and the compiler-distributer ends its task.

For the moment, the error contains (1) the description of the entire connected component, which rapidly becomes huge and is difficult to comprehend and (2) a shortest path between two colored nodes is given, calculated using the all-pairs shortest path algorithm of Floyd-Warshall [Flo62] which runs in $O(n^3)$ (where n is the size of the component).
A third option has been implemented as a debugging aid, which consists of the graphical presentation of the dependency graph using the open source toolkit `graphviz`[1].

If the atomic constraints are satisfied, all nodes having edges of infinite weight are merged together, and the coloring algorithm of section 4.3.2 can be applied to the graph. Two options are available : either the full shrinkage technique using MAX-MIN-`st` cuts, or the faster shrinkage resulting from theorem 13 can be applied. The implementation using MAX-MIN-`st` cuts uses a subroutine calculating max-flow with Tarjan's algorithm. This procedure was taken from the `GOBLIN` project (Graph Object Library for Network Programming Problems)[2], which is developed at the Universität Augsburg. However, we found the library unstable, and therefore implemented the faster shrinkage algorithm.

Upon termination of one of these coloring algorithms, some instructions may be left uncolored (cfr. section 4.2.2) because they are in a connected component without a fixed color. We implemented a greedy heuristics to approximate the underlying Minimum Makespan problem, thus coloring all

[1] `http://www.graphviz.org`
[2] `http://www.math-uni-augsburg.de/opt/goblin.html`

instructions and variables.

Synchronization code

Once all instructions are colored, the control flow graphs are altered to insert synchronization code as explained in chapter 4. In practice, the control flow graphs containing sequential code are modified in such a way that all instructions inside a basic block are of the same color. This involves splitting some basic blocks in different parts. At the end of each such part, synchronization code is inserted. However, the insertion of synchronization code is not fully terminated at this point. Recall that computation of the set of local variables to transmit involves the rather complex (i.e. time and memory consuming) process of calculating def-use chains. Since these chains are also needed to calculate the register allocation during code generation, this step is postponed until then.

Appendix F

The dSL compiler-distributer backend

In this appendix we discuss the implementation details of the frontend part of the compiler-distributer. The frontend is depicted in the global workflow of the compiler distributer which is represented in figure F.1.

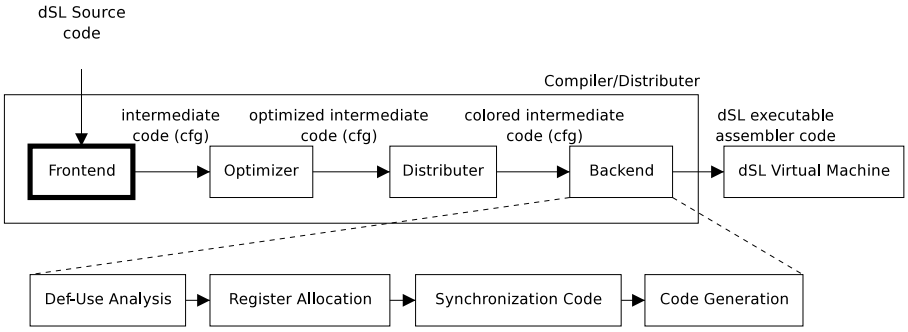

Figure F.1: The backend in the compiler-distributer workflow

Def-use analysis

The compiler-distributer backend proceeds by iterating over all control flow graphs. Recall that these control flow graphs are now colored, i.e. all instructions are assigned to a certain site, and that synchronization code is inserted. For each graph, three steps are performed :

1. Definition-use chains are calculated

2. Registers are allocated

3. Code is generated

The first task performed by the compiler-distributer backend consists in calculating the def-use chains. These are calculated by a backwards traversal of the basic blocks in the control flow graph. To each instruction i, we associate the set of instructions that use the value defined by i. At each program point p, a function U_p maps each variable x in the program to the set of instructions J that use x after p, such that x is not redefined between the point p and the instructions $j \in J$. This function is represented as a set of pairs (x, i), where x is a variable and i an instruction. For efficiency reasons, this set is not maintained at each program program point, but is stored only at the entry and exit of each basic block. Let $IN[B]$ and $OUT[B]$ denote the value of this function at, respectively, the entry and the exit from a basic block B in the control flow graph. The analysis proceeds as follows :

- Propagation of the information across a basic block B.
  Consider a 3-address assignment instruction $i \equiv$ `x := y op z`. Let the program points immediately before and after i be p' and p respectively. Let the U at the point p be U_p. Then we can define $U_{p'}$ as follows :
  $$U_{p'}(v) = \begin{cases} \emptyset & \text{if } v = x \wedge v \neq y \wedge v \neq z \\ \{i\} & \text{if } v = x \wedge (v = y \vee v = z) \\ U_p(v) \cup \{i\} & \text{if } v \neq x \wedge (v = y \vee v = z) \\ U_p(v) & \text{otherwise} \end{cases} \quad (1)$$
- Propagation of the information between basic blocks.
  For any basic block B, let the successor of that block be denoted by $succ(B)$. Then we have :
  $OUT[B] = \cup_{B' \in succ(B)} IN[B']$. (2)

This defines a backward dataflow analysis that can be solved iteratively. Starting with $IN[B] = OUT[B] = \emptyset$, a backwards traversal of all basic blocks calculates $IN[B]$ for each basic block B using equation (1). Next, $OUT[B]$ can be calculated using the newly calculated sets $IN[B]$, with equation (2). If any of the sets $IN[B]$ or $OUT[B]$ changed, a new iteration is performed.

Register allocation

Once the def-use chains are computed, the second task of the compiler-distributer backend can take over. This task consists of assigning a register number to all local variables. These variables are either local variables

introduced by the programmer, or may result from the simplification of complex expressions into 3-address code, as explained in section 5.1.3. The problem of assigning a minimum number of registers to these variables is formalized as the well known NP-Complete *minimum graph coloring problem* [GJ90] :

Definition 38 (Minimum graph coloring problem)
Given a graph $G(V, E)$, find a coloring of G, i.e. a partition of V into disjoint sets $V_1, V_2, \ldots, V_k$ such that each V_i is an independent set for G, minimizing the number of disjoint independent sets V_i. ♦

The formulation of register assignment as a graph coloring problem is introduced in [Bri92]. In this paper, Briggs defines a register interference graph, where nodes are variables and edges between nodes express the fact that two variables can not remain in the same register at the same time. With this formulation, minimizing the number of needed registers for a given control flow graph reduces to finding a minimum coloring of the underlying interference graph.

Two variables can not be in the same register if they are *live* at the same time. A variable x is live at program point p if there is a path in the control flow graph, starting from p, where x is used before it is optionally redefined [ASU86]. Liveness information can easily be obtained from the previously computed definition-use chains : if $U_p(x) \neq \emptyset$, then x is live at p. This information is recorded for each program location p when solving the data flow analysis presented above.

We can therefore construct the register interference graph, and use it to minimize the number of used registers. As stated above, this requires solving an NP-complete problem, and a heuristics is therefore used in the implementation. We implemented the heuristics from [Cha82], which is represented in figure F.2.

Note that the coloring heuristics may fail to find a coloring if there are nodes with more neighbors than available registers. In this case, additional code must be added which *spills* the local variable into memory. This operation removes some edges from the interference graph, and a coloring may be found [GBJL02]. This is not implemented in the compiler-distributer. We do not believe that there is a strong need for this operation since the dSL virtual machine counts 32 registers for each basic type (`BOOL, INT, DINT, LINT, ...`). Remark that because of this particular architecture, no edges exist between variables of different types, even if they are live at the same time.

```
let
  k      = number of available registers.
  G(V,E) = register interference graph.
  S      = empty Stack.


while (V not empty) {
   v = node in G with fewer than k neighbors;
   S.push(v,G);
   E = E \ { e | e = (x,v) };
   V = V \ {v};
}

while (S not empty) {
   (v,G) = S.pop();
   assign color c to v such that
          no neighbor exists in G with the same color.
}
```

Figure F.2: Heuristics for the minimum graph coloring problem

Completing the synchronization code

The next step performed by the backend completes the synchronization code. Recall that synchronization instructions are added to the end of basic blocks, if they have at least one successor which contains instructions of a different color (this transformation is performed by the distributer). Since these instructions are at the end of the basic blocks we can use the previously calculated sets $OUT[B]$ for each such a basic block, in order to complete the synchronization code with the set of variables that need to be send (cfr. section 5.1.4).

Code generation

Finally, the backend terminates by generating executable dSL virtual machine code. In contrast to normal compilers, where normally a single executable is generated, the backend generates as many executables as there are sites in the system. For each site, the output file contains

1. The description of all sites in the system.
2. The description of all types in the program.
3. The description of all constants used in the program (this may include

strings and floating point numbers).

4. The description of all global variables allocated on the current site.

5. The contents of all segments (i.e. basic blocks of the control flow graphs) known on this site. These segments may come from `WHEN`s, `METHOD`s and `SEQUENCE`s.

6. A list of triggers, which are triples (x, c, s), where x identifies a global variable, c identifies the segment containing the code which evaluates the condition of a `WHEN`, and s identifies the segment to execute if a raising edge is detected in the condition.

Appendix G

An Introduction to PROMELA

This overview is mainly based upon the document which can be found on the website of the *Spin* tool[1].

PROMELA programs consist of processes, message channels, and variables. Processes are global objects. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior, channels and global variables define the environment in which the processes run.

Executability

In PROMELA there is no difference between conditions and statements, even isolated boolean conditions can be used as statements. The execution of every statement is conditional on its executability. Statements are either executable or blocked. The executability is the basic means of synchronization. A process can wait for an event to happen by waiting for a statement to become executable. For instance, instead of writing a busy wait loop: `while (a != b) skip` one can achieve the same effect in PROMELA with the statement `(a == b)`.

Variables

Variables are used to store either global information about the system as a whole, or information local to one specific process, depending on where the declaration for the variable is placed. The scope of a variable is global if it is declared outside all process declarations, and local if it is declared within a process declaration.

[1] `http://www.spinroot.com`

Data types

The names `bit` and `bool` are synonyms for a single bit of information. A `byte` is an unsigned quantity that can store a value between 0 and 255. `short`s and `int`s are signed quantities that differ only in the range of values they can hold.

Array variables

Variables can be declared as arrays. For instance, `byte state[N]` declares an array of N bytes that can be accessed in statements such as `state[0] = state[3] + 5 * state[3*2/n]` where `n` is a constant or a variable declared elsewhere. The index to an array can be any expression that determines a unique integer value.

Declarations and assignments are always executable. Conditions are only executable when they hold.

Process types

The state of a variable or of a message channel can only be changed or inspected by processes. The behavior of a process is defined in a `proctype` declaration. The following, for instance, declares a process with one local variable `state`.

```
proctype A() {
      byte state;
      state = 3
}
```

The process type is named A. The body of the declaration is enclosed in curly braces. The declaration body consists of a list of zero or more declarations of local variables and/or statements. The declaration above contains one local variable declaration and a single statement: an assignment of the value 3 to variable state.

The semicolon is a statement separator (not a statement terminator, hence there is no semicolon after the last statement). PROMELA accepts two different statement separators: an arrow '`->`'and the semicolon ';'. The two statement separators are equivalent. The arrow is sometimes used as an informal way to indicate a causal relation between two statements.

Process Instantiation

A proctype definition only declares process behavior, it does not execute it. Initially, in the PROMELA model, just one process will be executed: a process of type `init`, that must be declared explicitly in every PROMELA specification. A typical init declaration looks as follows.

```
init
{       run A(); run B()
}
```

`run` is used as a unary operator that takes the name of a process type (e.g. A). It is executable only if a process of the type specified can be instantiated. It is unexecutable if this cannot be done, for instance if too many processes are already running. The `run` statement can pass parameter values of all basic data types to the new process.

Run statements can be used in any process to spawn new processes, not just in the initial process. Processes are created with the run statements. An executing process disappears again when it terminates (i.e., reaches the end of the body of its process type declaration), but not before all processes that it started have terminated.

Atomic and deterministic Sequences

By prefixing a sequence of statements enclosed in curly braces with the keyword `atomic` the user can indicate that the sequence is to be executed as one indivisible unit, non-interleaved with any other processes. If the sequence of statements is fully deterministic, and no statement can be blocking, then the keyword `d_step` may be used. `atomic` and `d_step` are powerfull features of the PROMELA language, because they allow to considerably reduce the size of the state space of the underlying model. We come back to this point further on.

Message Passing

Message channels are used to model the transfer of data from one process to another. They are declared either locally or globally, for instance as follows: `chan qname = [16] of { short, short }`.

This declares a channel that can store up to 16 messages, each of which is a pair of type short. Channel names can be passed from one process to another via channels or as parameters in process instantiations.

The statement `qname!expr1,expr2` sends the value of expression `expr1` together with the value of `expr2` to the channel that we just created, that is: it appends the pair of values to the tail of the channel. `qname?r1,r2` receives the message, it retrieves it from the head of the channel, and stores it in the variables `r1` and `r2`. The channels pass messages in first-in-first-out order. Constants can be used in the receive statement. In that case, the receive statement is executable only if the first message matches all corresponding constants.

Messages can also be taken inside the message queue by the statement `qname ?? <recv_args>`, where `recv_args` is a list of receive arguments. In that case, the first message in the queue that matches the arguments is taken. If none exist, the statement is not executable. If only variables are used in `recv_args`, the behavior is the same as `qname ? <recv_args>`.

The send operation is executable only when the channel addressed is not full. The receive operation, similarly, is only executable when the channel is non empty.

A predefined function `len(qname)` returns the number of messages currently stored in channel qname. Note that if len is used as a statement, rather than on the right hand side of an assignment, it will be unexecutable if the channel is empty: it returns a zero result, which by definition means that the statement is temporarily unexecutable.

Rendez-Vous Communication

So far we have talked about asynchronous communication between processes via message channels, declared in statements such as `chan qname = [N] of { byte }` where `N` is a positive constant that defines the buffer size. Rendezvous communication is obtained when the channel size is zero, that is, the channel port can pass, but can not store messages. Message interactions via such rendezvous ports are by definition synchronous. Rendez-vous communication is binary: only two processes, a sender and a receiver, can be synchronized in a rendezvous handshake.

Control Flow

Between the lines, we have already introduced three ways of defining control flow: concatenation of statements within a process, parallel execution of processes, and atomic sequences. There are four other control flow constructs in PROMELA to be discussed. They are case selection, repetition,

unconditional jumps, and inlining.

Case Selection

The simplest construct is the selection structure. Using the relative values of two variables a and b to choose between two options, for instance, we can write:

```
if
:: (a != b) -> option1
:: (a == b) -> option2
fi
```

The selection structure contains two execution sequences, each preceded by a double colon. Only one sequence from the list will be executed. A sequence can be selected only if its first statement is executable. The first statement is therefore called a guard.

Guards can be mutually exclusive, but they need not be. If more than one guard is executable, one of the corresponding sequences is selected non-deterministically. If all guards are unexecutable the process will block until at least one of them can be selected. There is no restriction on the type of statements that can be used as a guard. A process of the following type will either increment or decrement the value of variable count once.

```
byte count;

proctype counter() {
if :: count = count + 1
   :: count = count - 1
fi
}
```

The special guard `else` can be used, and is executable when all other guards are false.

Repetition

A logical extension of the selection structure is the repetition structure. We can modify the above program as follows, to obtain a cyclic program that randomly changes the value of the variable up or down.

```
byte count;

proctype counter() {
do :: count = count + 1
   :: count = count - 1
   :: (count == 0) -> break
od
}
```

Only one option can be selected for execution at a time. After the option completes, the execution of the structure is repeated. The normal way to terminate the repetition structure is with a break statement. In the example, the loop can be broken when the count reaches zero. Note, however, that it need not terminate since the other two options always remain executable.

Unconditional Jumps

Another way to break the loop is with an unconditional jump: the infamous goto statement.

```
do :: count = count + 1
   :: count = count - 1
   :: (count == 0) -> goto enddo;
od
enddo: ...
```

Inlining

Inlining is a pure syntactical addition which allows one to better organize the PROMELA specification. By means of the keyword `inline`, a piece of the specification can be named and substituted in any part of the specification that follows it. Just like functions in common programming languages, parameters may be passed, but no recursion is allowed. Inlining is performed during macro-preprocessing.

```
inline behavior(my_id) {
    a_channel ! my_id;
}

proctype proc_1() {
    behavior(1);  // a_channel ! 1;  is substitued here
}
proctype proc_2() {
    behavior(2);  // a_channel ! 2;  is substitued here
}
```

Assertions

Statements of the form `assert(any_boolean_condition)` are always executable. If the boolean condition specified holds, the statement has no effect. If, however, the condition does not necessarily hold, the statement will produce an error report during verifications with Spin.