UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des sciences.
Département d'informatique.

On the verification of $_{d}$SL, a language to design distributed industrial control systems.

Alexandre Genon

Mémoire en vue de l'obtention du DEA en sciences.

Année academique 2003-2004.

Contents

Chapter 1

Introduction

Industrial process control goes hand in hand with distributed systems. This is due to the physically distributed nature of the environment that is continuously controlled through various devices such as sensors and actuators. Development of such distributed systems is a complicated task, even for experienced programmers. The burden of combining the physical complexity of the process, 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.

To simplify the work of the distributed systems designer, classical solutions do exist (CORBA, DCOM, EJB,...) which handle the communication aspects and allows the programmer to concentrate on the functionality aspects of the system. Unfortunately, these solutions are generally quite heavy, and completely hide all the communications aspects, making the monitoring of such systems difficult.

In this paper, we first introduce $_{\text{d}}$SL which is a simple imperative and event driven language designed to program distributed industrial control systems. The advantage of $_{\text{d}}$SL is to provide a transparent code distribution using low level mechanisms. Therefore, most of the time the $_{\text{d}}$SL programmer can ignore all the communication aspects between controllers of the distributed systems. Moreover, by the simplicity of the distribution mechanisms, the behavior of the synthesised distributed system can be formally modeled and easily monitored - which is a main concern for this kind of systems.

Industrial systems are often criticals and simple errors may lead to great losses. To have a complete solution to design industrial systems, we must include the possibility to *verify* $_{\text{d}}$SL programs. Indeed, using verification techniques on a given program, we can be more confident about its use in critical environments. The problem with verification is that it is a complicated task and it is sometimes unrealisable in practice. This paper presents two techniques that could be used during the verification of $_{\text{d}}$SL. The first technique is based on a result which links various distributions of the $_{\text{d}}$SL program to analyse. The second technique is the *slicing*. Slicing is a technique used to reduce the size of a program by tearing off some instructions independent from the property to verify. A slicer for $_{\text{d}}$SL is implemented and allows us to perform verification on subprograms containing only necessary instructions.

Plan of this paper. The paper is organised as follows. The first chapter presents the $_{\text{d}}$SL language. We detail related proposals and justify the advantages of our solution. The syntax and semantics are also presented. Chapter two presents a more detailed analysis of the concept of distribution in $_{\text{d}}$SL. A theorem which relates different distributions is presented. This theorem is an important result for the verification.

Chapter three introduces the technique of slicing, commonly used in verification. We study how to implement a slicer for ${}_{\mathrm{d}}$SL. The final chapter presents the conclusions of this work.

Acknowledgement We would like to thank Prof. Raymond Devillers for the careful review of this document. We also would like to thank Bram De Wachter and Cedric Meuter for their insightful comments. Finally, we would like to thank all the members of the jury, i.e. Prof. Martine Labbé, Prof. Thierry Massart and Prof. Jean-Francois Raskin.

Chapter 2

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

In this chapter, we present the language ${}_{\mathrm{d}}$SL, used to design industrial control systems. We first give the motivations for this new language and compare it with other approaches. An informal introduction to ${}_{\mathrm{d}}$SL concepts and an example are also given. To give a more formal but still concise view of ${}_{\mathrm{d}}$SL, we need to pass through a stripped version of ${}_{\mathrm{d}}$SL, named ${}_{\mathrm{d}}\mathrm{SL}_{\diamond}$. We can automatically translate a ${}_{\mathrm{d}}$SL program to a ${}_{\mathrm{d}}\mathrm{SL}_{\diamond}$ program. Then, we present the formal syntax and semantics for this reduced version of ${}_{\mathrm{d}}$SL. This chapter is a summary of [WGMM04].

2.1 Motivations and other approaches

The problem of distributing applications that control reactive systems has been studied for many years now and several interesting observations on these works shaped the design of ${}_{\mathrm{d}}$SL. In particular, this problem has been studied conceptually in the world of process algebras and defined as a correctness preserving transformation of a *centralised* specification into a semantically equivalent distributed one. (e.g. for bisimulation equivalence [Mil89], see [Mas92, BL95]). It has also been studied on various types of labelled transition systems ([CMT99, Mor99, SEM03]).

These works solved part of the problem. However, contrary to other programming languages, the notion of variables does not exist in usual process algebra and these solutions had therefore to be extended. Work has also 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. Unfortunately, in our opinion, the distribution of synchronous languages while preserving the semantics, suffer from a performance problem which, in practice, may not be acceptable.

Indeed, the synchronous programming scheme found in the synchronous languages supposes that time is defined as a sequence of *instants*. To preserve determinism, these languages use the concept of synchronous broadcast [Ber89, BB91] when several processes are composed in parallel. This implies that parallel branches in the high level description can be transformed into sequential deterministic code. The distribution of such programs, for example in Esterel [Gir94], may suffer from severe performance penalties :

1. because the *instants* must be respected, requiring a strong resynchronisation scheme

2. and because the distribution is applied on the determinised sequential code.

For example, the distribution of Esterel described in [Gir94] can be summarised in 4 steps : (1) the *centralised* program (after being compiled in a single threaded sequential code) is duplicated on all participating sites; (2) the instructions that are not relevant to a given site are removed; (3) for data that is accessed on one site, but calculated on another, communication messages are inserted, and (4) synchronisation messages are inserted to preserve the global instants which can slow down the whole system. Remark that since the initial code is sequential, this solution suffers from the lack of parallelism (there are some ways to achieve higher concurrency such as weak synchronisation but that does not preserve safety properties [Gir94]). For Signal, the situation is very similar [Aub97].

This strong synchronisation has several undesirable drawbacks in an industrial environment. First of all, to keep all processes in pace, numerous messages need to be exchanged at each global instant. Secondly, all participating processes have to advance at the speed of the slowest process. Finally, the failure of one of the processes makes the whole system deadlock. To the best of our knowledge, the synchronous approach has no answer to these shortcomings.

These observations make us believe that, although perfectly suitable for tightly coupled homogeneous systems and having the benefit of simplicity when it comes to specifying a controller, the simplicity of the synchronous approach is too costly in terms of performance when applied to loosely coupled heterogeneous systems. Moreover, in pratice, the strong synchronisation of all processes is only rarely needed and must therefore not be used by default.

To avoid these drawbacks, dSL rejects the synchronous product [Mil81] used in the above languages, and adopts an asynchronous composition of instantaneous (*atomic*) code and asynchronous (*sequential*) code. Asynchronous composition is therefore the keyword in dSL's design. The instantaneous code uses an event driven scheme and, for a given component, must be able to run without any synchronisation 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. But this cooperation may introduce deadlocks. These assumptions, of course, imply some restrictions on code and have consequences on the way data values are transmitted between distributed processes, as explained in section 2.4.

For the distributed execution of the *sequential* code, several models are proposed in the literature. These models can be divided into two sets based on the way they achieve data locality : either move data, or move the execution. Many systems have been studied that use the first solution, such as Distributed Shared Memory systems [NL91]. These systems, although offering a transparent distributed environment, suffer from undesirable border effects that make them unusable in an industrial environment. The need to replicate data to make such systems work in a performance responsible way [HHG99], may cause thrashing (i.e. two, or more, processes competing for exclusive access to a given variable, resulting in high communication traffic and almost no productiveness) or false sharing (i.e. two, or more, variables used by different processes, but being in the same page cause unnecessary communication traffic) and these systems therefore do not guarantee stable performance as observed in [NL91]. Secondly, since data moves around, the supervision of such systems and its error-recovery - both mandatory features in industrial applications - may become too complicated [MP97] on dSL's target hardware.

For these reasons, dSL uses the second solution, which consists of moving the execution to the data, a concept known as process or thread migration [Esk90]. In this concept, 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]. In our system, all instructions and global variables are statically assigned to the participating sites and thread migration, decided at compile time, is used to obtain data access locality. The benefits are twofold: (1) following the state of the system is very easy, and (2) all communication and synchronisation messages can be statically calculated, resulting in a predictable execution. However, we lose the benefits of dynamic load balancing and fault tolerance since the migration policy used in dSL is static.

The design of dSL can thus be synthesised as 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.

2.2 Informal presentation of dSL

A control system is generally a distributed system made of one or several communicating *sites* each of which can be either a supervisor (typically a computer maybe with a user interface) or a programmable controller (called automata from here on) which are connected through sensors and actuators to the industrial equipment to control. The industrial system to control is seen as the *environment* of the system. The various sites are interconnected through a network and transmit data through FIFO queues.

dSL [1] is both a programming language used to design industrial control systems and a programming environment, mainly a compiler which synthesises the distributed control systems. It is a simple imperative language with static variables; a variable can be (1) internal to the program (2) linked to an input (sensor) or (3) linked to an output (actuator). dSL is event driven, i.e., an event is specified by the change in some variable value. For instance, `when x >= 0 then run_motor1();` will trigger `run_motor1()` every time the variable x switches from a negative to a positive value. Also note that dSL has limited Object-Oriented features. Variables and methods are therefore structured into object definitions. We will not detail this aspect further in this paper.

At first glance, a dSL program seems to be designed to control a centralised (i.e. non distributed) environment. A dSL program is written as if the entire environment can be accessed without the need for explicit communication or synchronisation primitives (we shall see that some restrictions are imposed to apply this principle).

When the designer has written the dSL program he must also fill in a *localisation table* to specify the physical localisation (site) of each I/O. The dSL compiler can then automatically distribute the code among the sites (supervisors or automata), trying to minimise the needed communications[2]. Figure 2.1 shows the structure of a dSL implementation.

It is clear that this approach has many benefits such as (1) maintainability (only one language is used) (2) flexibility (changing an actuator / sensor from one site to another does not imply changes on the program),(3) simplicity (since communication/distribution is done implicitly, the programmer does not need to come up with synchronisation schemes to handle particular tasks).

Example 2.1 - dSL example

To illustrate the dSL concepts, let us give, in figure 2.2 a simplified example of a program that receives input values through an input variable `temp`, linked to a temperature sensor and switches on (off) a `heater` when the

[1] dSL is the successor of the language SL (Supervision Language) developed by the Macq Electronique company, Belgium that was originally designed for controlling and supervising industrial processes

[2] The problem to minimise the number of communications is hard; it is not the subject of this paper

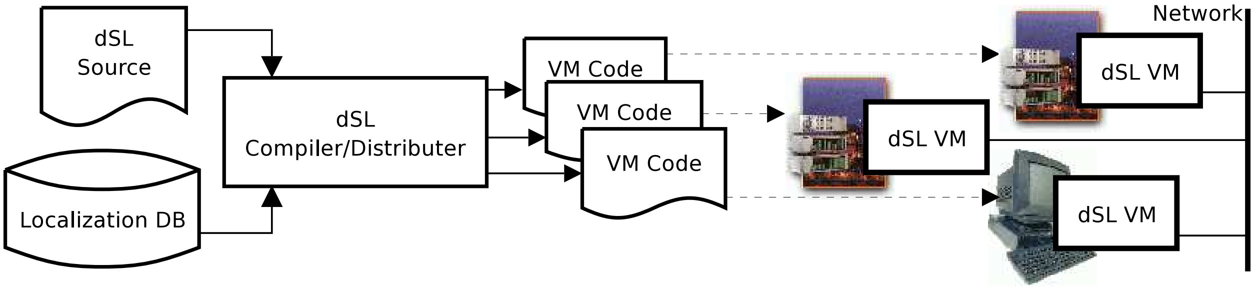

Figure 2.1: How a dSL program is implemented.

temperature is below 0 ^{o}C (above 20 ^{o}C). Two indicators in a control panel (`led` and `alarm`) are also updated following the state of the heater and the number of times the heater is turned from one state to the other, respectively. The program will be used in a physical configuration with 2 sites; but this is not directly mentioned in the program. The dSL program of figure 2.2 corresponds to this simple control system.

Atomic and sequential code

The design of dSL has been dictated by the execution paradigm used in the world of industrial process control that requires a immediate reaction to events and their instantaneous treatment. In practice, by default, this forbids any hidden synchronisation delaying execution and in particular synchronisation which implies inter-site communications (through a relatively slow network). A clear way must therefore exist to express that inter-site synchronisation is allowed. Hence dSL distinguishes between

- *atomic* or *event-driven* code which is executed in an atomic manner and is seen as instantaneous and therefore cannot be distributed and
- *non atomic* or *sequential* code which can be distributed and uses inter-site communications to synchronise or transfer values between sites.

By default, the code attached to a `WHEN` is *event-driven* and therefore must be local to a given site. To relax this constraints, two mechanisms have been defined: (1) the `LAUNCH` instruction allows to start *sequential* code, and (2) *event-driven* code may use *tilded* variables (also called *asynchronous variables*) to access the last locally known value of a variable. For instance, the instruction `y := ~x` assigns to the local variable `y` the last value received in the local site of the distant variable `x`, even if the value of `x` has changed since then. More precisely, if variable `y` is local to site i, `~x` is a copy of variable `x` local to site i, which is updated asynchronously every time `x` is modified. *Tilded* variables must be used with care : only when the exact value of a variable which cannot be local is not needed (e.g. temperature which evolves slowly) or if the program is built such that it is known that the *tilded* value is equal to the real one.

In dSL, *sequential* code is defined through the `SEQUENCE` instruction. A `SEQUENCE` cannot have more than one instance executed simultaneously. A `SEQUENCE` can have local variables and, when needed, these variables can be transmitted from site to site to ensure a coherent execution of the `SEQUENCE`. `SEQUENCE`s are called through the use of `LAUNCH`s, note that these `LAUNCH`s must be such that it can be atomically executed.

```
GLOBAL_VAR
  led         : output BOOL;
  heater      : output BOOL;
  alarm       : output BOOL;
  temp        : input BOOL;
  maintenance : output BOOL;
  control     : output INT;
END_VAR

SEQUENCE set_heater(state : BOOL)
(* Make the led's state correspond to
   the heater's action *)
   control := control+1;
   IF control == 1000 THEN
      control := 0;
      maintenance := TRUE;
   END_IF;
   led := state;
   heater := state;
END_SEQUENCE

WHEN maintenance THEN
  alarm := TRUE;
(* do something *)
END_WHEN

WHEN temp < 0 THEN
(* Turn on the heater *)
   launch set_heater (1);
END_WHEN

WHEN temp > 20 THEN
(* Turn off the heater *)
   launch set_heater (0);
END_WHEN

SEQUENCE init()
   control := 0;
(* Initially turn the heater off *)
   LAUNCH set_heater(0);
END_SEQUENCE
```

Figure 2.2: simple dSL program

i.e. all variables used in parameters must be from the same site. Note that `LAUNCH` can also be used to start the execution of a sequence located on another site. The `init` sequence is a special sequence without parameters that is launched when the program is started.

2.3 dSL syntax

We now present the syntax for dSL. A dSL program is composed by a list of global variable declarations, a list of `WHEN`'s, a list of `METHOD`'s, and a list of `SEQUENCE`'s. A variable declaration gives the name of a variable, its type (`int` or `bool`), and its *kind* (`internal`, `input`, or `output`). `METHOD`'s and `SEQUENCE`'s can also have local variables. `WHEN`'s are composed of a *triggering* condition, and a list of instructions (i.e. the body of a `WHEN`) that must be executed when the triggering condition switches from false to true. `METHOD`'s are entry points that may be called synchronously or asynchronously. A `METHOD` must be located on a single site. `SEQUENCE`'s can only be called asynchronously. But a `SEQUENCE`'s execution can be distributed among multiple sites. Asynchronous calls can be done through the use of the `LAUNCH` keyword. A grammar for dSL can be found in appendix A. Moreover, we add the following syntactic restriction : `while` loops are forbidden in `WHEN`'s body. This restriction is needed to bound the execution time of each `WHEN`. A program that respects this grammar and the latter restriction is called a *well-formed* dSL program.

From now on, we will use the following functions :

- $Whens(P)$ denotes the set of `WHEN`s appearing in the program P.
- *Seqs*(P) denotes the set of `SEQUENCE`'s appearing in the program P.
- *Meths*(P) denotes the set of `METHOD`'s appearing in the program P.

- $<_W(P)$ denotes the order in which the WHENs of program P are declared. This order will be used to determine the order in which the WHENs will be processed.
- $Cond(w)$ denotes the triggering condition of the WHEN w.
- $Body(e)$ denotes the body of the WHEN, SEQUENCE or METHOD e
- $Var_A(P)$ denotes the set of *all* variables appearing in the program P (i.e. the global variables and the local variables and parameters for all METHOD's, and SEQUENCE's).
- $Var(P)$ denotes the set of non tilded, global 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.
- $Var(b)$ returning the set of non tilded, global variables appearing in the body b (body of a WHEN, METHOD, or SEQUENCE).
- $Var(e)$ returning the set of global variables (*tilded or not*) appearing in expression e.
- $Var(w)$ returning the set of global variables appearing in the WHEN w (i.e. $Var(w) = Var(Body(w)) \cup Var(Cond(w))$).
- $<_V(P)$ denotes the order in which the global 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.
- $OldCond(W) = \{old\_cond_w | w \in W\}$, for all $w \in W$, old_cond_w is a new *id* corresponding to the previous evaluation of $Cond(w)$. This is used when considering the triggering of w.
- $\tilde{X}$ denotes the set of tilded variables corresponding to X, $\tilde{X} = \{\tilde{x} | x \in X\}$

2.4 Preliminary definitions

Now that we have given a syntax for the language dSL, we can give its semantics. In order to keep the semantics simple, we present only a subset of the dSL language, called here the dSL$_\Diamond$ language. Section **??** presents a translation from a dSL program to a dSL$_\Diamond$ program. But, before detailing the translation from dSL to dSL$_\Diamond$, we need to define some concepts of dSL (and dSL$_\Diamond$). We first define some concepts on SEQUENCE's that will be useful for their translations. Then, we introduce the concept of distribution in dSL.

Definitions

This section presents some preliminary definitions. To perform the translation from dSL to dSL$_\Diamond$, we need to identify precisely what is an instruction in a sequence. The next definition details the set of instructions in the body of a SEQUENCE.

Definition 2.1 - Set of instructions

Let s be a SEQUENCE $\in Seqs(P)$ of a well-formed dSL program P. We define $Instr_s(s)$ as the set of instructions of s. More formally :

- $Instr_s :\ Seqs(P) \rightarrow 2^{\Sigma^* \times Id_s}$.
- Let $s \equiv$ `SEQUENCE` id ($param\_decl$) var_decl $instruction\_list$:
  $Instr_s(s) = \{(\omega, id)\}$. Where ω gives the remaining part of the body of s, starting from this instruction.

$Instr_s(s)$ returns the set of all instructions in s's body. Here, an instruction i is encoded as couple (ω, id). An instruction is not only identified by its code, but also by its position in the `SEQUENCE`'s body. id returns the id of the sequence. This is needed to distinguish between two instructions in two different sequences that have the same ω.

We can define $Instr_s(P)$ as $\cup_{s \in Seqs(P)} Instr_s(s)$. From now on, we will use the following function : $Var(i),\ i \in Instr_s(P)$ that returns the non tilded, global variables appearing in instruction i. Now that we have the set of instructions of a `SEQUENCE`, we may define a successor relation. This relation gives a sketch of the control flow of a sequence and will be used to distribute the execution among different sites.

Definition 2.2 - Instruction's successor, first and last instructions of `SEQUENCE`
Let P be a well-formed dSL program, we define the following functions for P :

- $Succ(S, i) : Seqs(P) \times Instr_s(P) \mapsto 2^{Instr_s(P)}$ that returns, for a given instruction $i = (\omega, id)$ in `SEQUENCE` S the set of immediate successor instructions of i.
- $First(S) : Seqs(P) \mapsto Instr_s(P)$ that returns the first instruction of `SEQUENCE` S.
- $Last(S) : Seqs(P) \mapsto 2^{Instr_s(P)}$ that returns the last instructions that may be executed by `SEQUENCE` S.

These functions are defined using the following attribute grammar. Note that in this grammar, we suppose that each instruction can be uniquely identified in the dSL program. For example, we can use the (ω, id) encoding presented above.

i_list	$\longrightarrow$	$i; i\_list_1$	$i\_list.succ := i\_list_1.succ \cup i.succ$
			$\cup_{i' \in i.last}\{(i', i\_list_{1\,1}.first)\}$
			$i\_list.first := i$
			$i\_list.last := i\_list_1.last$
i	$\longrightarrow$	$if \mid while \mid assign \mid$	
		$sync\_call \mid async\_call$	
if	$\longrightarrow$	`IF` e `THEN` i_list_1	$if.succ := i\_list_1.succ \cup i\_list_2.succ \cup$
		`ELSE` i_list_2 `END_IF`	$\{(if, i\_list_1.first), (if, i\_list_2.first)\}$
			$if.first := if$
			$if.last := i\_list_1.last \cup i\_list_2.last$
$while$	$\longrightarrow$	`WHILE` e `DO`	$while.succ := i\_list.succ \cup \{(i\_list.last, while)\} \cup$
		i_list `END_WHILE`	$\{(while, i\_list.first)\}$
			$while.first := while$
			$while.last := \{while\}$
$\forall \mathcal{I} \in \{assign,$			$\mathcal{I}.succ := \emptyset$
$synch\_call, asynch\_call\}$			$\mathcal{I}.first := \mathcal{I}$
			$\mathcal{I}.last := \{\mathcal{I}\}$

In this grammar, 3 attributes are propagated for each instruction :

- $succ \subseteq instruction \times instruction$, which calculates the immediate successors
- $first \in instruction$, indicates the first instruction of an instruction block.
- $last \subseteq instruction$, indicates the last instructions of an instruction block.

Note that, apart from *while* and *if* instructions, all other instructions have at most one successor in the *succ* attribute. For a given `SEQUENCE` s defined by

sequence_decl $\longrightarrow$ `SEQUENCE` *id* `(` *param_decl* `)` `begin_var` *var_decl* `end_var` *instruction_list* `END_SEQUENCE`

the immediate successors function is defined by *instruction_list.succ*. the first instruction function is defined by *instruction_list.first*. And the last instructions function is defined by *instruction_list.last*.

Maximal distribution

As we have already seen, the main difference between dSL and any common imperative programming language results from its distributed characteristics. Indeed, the behavior of dSL program depends on the distribution of its variables. However, the *maximal distribution* of a dSL program can be defined. It expresses the most liberal configuration in which the dSL program could ever run. Indeed, due to the *atomic* code feature, some instructions and hence variables must be kept on a single site. We will show that this *maximal distribution* allows the most possible behaviors. Therefore, verifying safety properties on this distribution will induce safety for any other distribution. Note that this is not the case for liveness properties.

Before we can define the *maximal distribution*, we first need to define what a distribution is. Intuitively, a distribution of a well-formed dSL program is a partition of the set of its global variables respecting the atomic constraints imposed by `WHEN`s, `METHOD`'s, and instructions in `SEQUENCE`'s. That is, if two global variables appear un-tilded in the same `WHEN`, `METHOD`, or instruction, then they must be localised on the same execution site. With these constraints, we are sure that every global variable will be localised on a unique site and that every `WHEN`, `METHOD`, and instruction (in `SEQUENCE`'s) will be executed atomically.

Definition 2.3 - Distribution of a well-formed dSL program
A distribution of a well-formed dSL program P is a partition $D = \{V_1, V_2, ..., V_n\}$ of $Var(P)$ such that

- $\forall w \in Whens(P), \exists V \in D, (Var(Cond(w)) \cup Var(Body(w)) \subseteq V)$. We should be able to localise each `WHEN` on a single site.
- $\forall m \in Meths(P), \exists V \in D, (Var(Body(m)) \subseteq V)$. We should be able to localise each `METHOD` on a single site.
- $\forall i \in Instr_s(P), \exists V \in D, (Var(i) \subseteq V)$. We should be able to localise each instruction in a `SEQUENCE` on a single site.

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

Note that, following this definition of distribution, we have that two `WHEN`'s that share (directly or indirectly) a same global variable will be localised on the same site.

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 distributions.

Definition 2.4 - 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 program P. We say that distribution D' is a refinement of distribution D, noted $D \preceq D'$ if

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

We can now formally define the *maximal distribution* of a well-formed $_d$SL program, which is the most refined distribution.

Definition 2.5 - Maximal distribution of a well-formed $_d$SL program
The maximal distribution of a well-formed $_d$SL program P is the distribution D_{max} such that

$$\nexists D \in \mathcal{D}_P \neq D_{max} : D_{max} \preceq D$$

Note that this maximal distribution always exists. Moreover, we can easily prove that this distribution is unique. Using this notion of maximal distribution, we may *locate* the different instructions of a `SEQUENCE`. This is the purpose of the next definition.

Definition 2.6 - Partioning the sequential instructions
Let $D_{max} = \{V_1, \cdots, V_m\}$ be the maximal distribution of a well-formed $_d$SL program P. We define the partition of the set of sequential instructions $Instr_s(P)$ as a partition $I = \{I_1, \cdots, I_n\}$ of $Instr_s(P)$ such that :

- $i \in I_j, j \in [1, m]$ iff $Var(i) \subseteq V_j$. This instruction should be localised on the same site as the variables that appear non tilded in it.
- $i \in I_j, j > m$ iff $Var(i) = \emptyset \ \wedge \ I_j = \{i\}$. There is no global variable that appears non tilded in this instruction. Thus, it can be localised on any site. We model this by putting this instruction in its own partition. When examining a $_d$SL program (i.e. for the translation to $_d$SL$_\Diamond$), we cannot make any hypothesis about its future localisation.

Moreover, we define the function $\text{LOC}\ (i) \stackrel{def}{=} j | I_j \ni i$

Note that some of the I_k sets may be empty (i.e. the variables in V_k are not used in any sequence). It is supposed that these sets are removed from I to have a correct partition.

2.5 From $_d$SL to $_d$SL$_\Diamond$

In this section, we intuitively describe the syntactical transformation from a full featured $_d$SL program to a simplified $_d$SL$_\Diamond$ program. As said earlier, $_d$SL$_\Diamond$ is a subset of the $_d$SL language. In this subset, we make the following restrictions: (1) tt METHODs are supposed to be inlined, which implies that recursive calls are forbidden; (2) since no recursion is allowed, all variables (even the parameters) can be considered to be declared globally; (3) only boolean variables are considered; (4) as we will see in this section, `SEQUENCE`s and `LAUNCH`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 will be given for `SEQUENCE`s and `LAUNCH`s We successively suppress `SEQUENCE`s, `LAUNCH`s and `METHOD`s as shown in figure 2.3. Indeed, the first transformation will translate `SEQUENCE`s by introducing new variables, `WHEN`s and `METHOD`s. Then, all asynchronous calls to

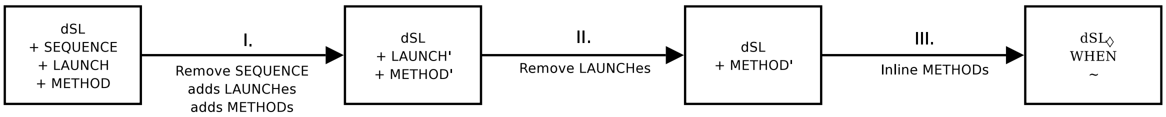

Figure 2.3: 3-step transformation of $_{\mathrm{d}}$SL

METHODs (i.e. LAUNCHs) are translated using new variables and WHENs conditionned on these variables. Finally, we remove synchronous calls to METHODs by inlining them. Note that the semantics of SEQUENCEs and METHODs is defined by this transformation. As said earlier, a SEQUENCE is a piece of code whose execution can be distributed among various sites. We use the LAUNCH keyword to start the execution of a SEQUENCE. Note that, thanks to this LAUNCH mechanism, a given site can execute a SEQUENCE whose first instruction is located on another site. A METHOD is a piece of code whose execution must be atomic and, thus, located on a single site. METHODs can be synchronously called, but only from the same site. Thanks to the LAUNCH mechanism, we can also asynchronously call a METHOD from any other site.

To describe this translation, we must be able to locate each instruction. For this, we use the maximal distribution. Thus, the behavior of a SEQUENCE will be ruled by this maximal distribution.

Suppressing sequences

A SEQUENCE is tranformed into a list of WHEN's and METHOD's. Since a SEQUENCE's execution may be distributed among various sites, when a given site executing a SEQUENCE cannot execute the next instruction (because it is located on another site), it sends an *hand-over* message to the site owning this instruction. When this site receives this message, it continues the execution of this SEQUENCE.

Thus, for each instruction of a SEQUENCE, we must have a separate WHEN that will run this instruction. Moreover, we have a METHOD that will trigger this WHEN when we reached this instruction. We simply call this METHOD when we want to run the corresponding instruction. If the previous instruction is located on the same site than the instruction to run, then we simply perform a METHOD call. Else, we must use the LAUNCH mechanism. As said earlier, we use the maximal distribution to know the localisation of each instruction. Note that we must also transmit the local variables and parameters of the SEQUENCE before executing the next instruction. We also have the restriction that only one instance of a SEQUENCE may be run at any time. And we must take care of this restriction when considering the triggering of the WHEN for the first instruction.

A formal description of this translation can be found in appendix B.

Removing LAUNCHes

Once all SEQUENCEs are translated into WHENs and METHODs, we must remove all LAUNCHs keywords. For this, we use the $\sim$ mechanism. For each launch instruction, we introduce a new variable. And we replace this LAUNCH instruction by an assignement to this variable. A WHEN, conditionned on a tilded copy of this variable is triggered and simply performs another assignment that will trigger the WHEN for the first instruction of the SEQUENCE.

A formal description of this translation can be found in appendix B.

Inlining synchronous calls to `METHODS`

We now have a $_{\mathrm{d}}$SL program P containing only synchronous calls to methods. As these calls are synchronous, the method will be directly executed when the call instruction is run. Thus, we could simulate a call by inlining the code of the method at the place of the call. Remember that recursion is forbidden. Thus, we can transform every local variable and parameter of this method as a global variable. For each call c of a method M, we first initialise the latter variables with their actual parameters. Then the code of the called method is inserted. Once we have treated all calls, we may remove the methods from P and the resulting program P' is a $_{\mathrm{d}}\mathrm{SL}_\Diamond$ program. Indeed, this program contains only `WHEN`'s and $\sim$.

`init SEQUENCE`

In a $_{\mathrm{d}}$SL program, we have a special `SEQUENCE`, called the `init SEQUENCE` This `SEQUENCE` is launched when the $_{\mathrm{d}}$SL program is started. In $_{\mathrm{d}}\mathrm{SL}_\Diamond$, we translate this `SEQUENCE` as any other `SEQUENCE`, except that the `WHEN` for the first instruction is conditioned on a special input variable. It is supposed that this variable is set to `true` by the environment or the virtual machine when the $_{\mathrm{d}}$SL program is started.

2.6 $_{\mathrm{d}}\mathrm{SL}_\Diamond$'s semantics

Using the transformations described in the previous section, we can transform a full-featured $_{\mathrm{d}}$SL program into a $_{\mathrm{d}}\mathrm{SL}_\Diamond$ program. We introduced this transformation because it is easier to describe the semantics of $_{\mathrm{d}}\mathrm{SL}_\Diamond$. From now on, we only consider $_{\mathrm{d}}\mathrm{SL}_\Diamond$ programs. In order to define the $_{\mathrm{d}}\mathrm{SL}_\Diamond$'s 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 update of the variable *id* to all execution sites,
- `MSG`, modeling the treatment of messages from the Fifo channel.

Preliminary definitions

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

Definition 2.7 - Labelled transition system
A labelled transition systems 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 symbols called the alphabet, with $\tau \notin \Sigma$ (τ is the internal 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 $q' = q_n$. If $w = \epsilon$, $q \xrightarrow{w} q'$ iff $q = q'$.

Then, given a well formed $_d\mathrm{SL}_\Diamond$ program P and a distribution D of P, we need to define the distributed context 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. This distributed execution context can be defined as follows.

Definition 2.8 - Distributed execution context
Given a $_d\mathrm{SL}_\Diamond$ program P, and a distribution $D = \{V_1, V_2, ..., V_n\}$ of P, we define the distributed execution context 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 $W_i = \{w \in Whens(P) | Var(w) \subseteq V_i\}$. In the following, we will call each (V_i, W_i) a local execution context and will 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. The first two functions defined hereafter construct lists of extended instructions corresponding respectively to the input and the output of the variables in a given set V.

Definition 2.9 - Input sampling, output writing
Given a set of variables V, we define $Sample(V, <_V)$, respectively $Write(V, <_V)$, as a list of instructions performing the input, respectively output, of the input (resp. output) variables in V in the order given by $<_V$, as follows:

$$\begin{array}{rcl} Sample(V, <_V) & = & \mathtt{INPUT(v_1); INPUT(v_2); ...; INPUT(v_k)} \\ Write(V, <_V) & = & \mathtt{OUTPUT(v_1); OUTPUT(v_2); ...; OUTPUT(v_k)} \end{array}$$

where $\forall i \in [1..k-1], v_i <_V v_{i+1}$ and $\bigcup_{i \in [1..k]} \{v_i\} = V$.

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

Definition 2.10 - 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 $\forall i \in [1..|W|]$:

$$\omega_i = \mathtt{IF}\ (Cond(w_i)\ \mathtt{AND\ NOT}\ old\_cond_{w_i})\ \mathtt{THEN}\ old\_cond_{w_i} := \top; Body(w_i)\ \mathtt{ELSE}\ old\_cond_{w_i} := \bot\ \mathtt{END\_IF}$$

and where $\forall i \in [1..|W|-1], w_i <_W w_{i+1}$ and $\bigcup_i \{w_i\} = W$.

Structural operational semantics

Informally, the behavior of a $_d$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 S_i governs a set of variables in V_i 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 value of the distant variables.

Execution of each S_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 and incoming messages are processed and (3) *Output* : variables linked to the outputs control the physical state of the devices they are connected to.

The formal structural operational semantics for a well formed $_d$SL$_\Diamond$ program P, w.r.t. a distributions D is given below as a labelled transition system whose visible actions are updates to/from the environment of the I/O variables. Let us first define a global state of a $_d$SL$_\Diamond$ program.

Definition 2.11 - Global state of a $_d$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:

$$G \equiv ((\omega_1, \nu_1, \phi_1), (\omega_2, \nu_2, \phi_2), \ldots, (\omega_n, \nu_n, \phi_n))$$

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.
- $\nu_i : (V_i \cup \widetilde{Var(P)} \cup OldCond(W_i)) \mapsto \{\top, \bot\}$ is a valuation function for:
  - the global variables owned by process i (i.e. V_i).
  - the tilded (local) copies of all variables (i.e. $\widetilde{Var(P)}$).
  - the variables containing the old value for the conditions of each `WHEN` owned by process i (i.e. $OldCond(W_i)$).
- $\phi_i \in (Var(P) \times \{\top, \bot\} \times \mathbb{N})^*$ is the receiving communication channel of process i. ϕ_i may contain messages of the form (x, v, i) where x is the variable updated, v its new value, and i the source of this message (i.e. the process owning x).

We will note $\mathcal{G}_D^P$ the set of global states of a $_d$SL$_\Diamond$ program P, given a distribution D.

We can now introduce the semantical rules which will provide the transition relation of the labelled 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, contrary to synchronous languages like Esterel or Lustre, $_d$SL$_\Diamond$ (and, thus, $_d$SL) has an asynchronous semantics

[Interleaving]

$$\frac{(E_D^P)_i \vdash (\omega_i, \nu_i, \phi_i) \xrightarrow{a} (\omega_i', \nu_i', \phi_i')}{E_D^P \vdash ((\omega_1, \nu_1, \phi_1), ..., (\omega_i, \nu_i, \phi_i), ..., (\omega_n, \nu_n, \phi_n)) \xrightarrow{a} ((\omega_1, \nu_1, \phi_1), ..., (\omega_i', \nu_i', \phi_i'), ..., (\omega_n, \nu_n, \phi_n))}$$

$$\forall a \in \{\tau\} \cup \Sigma$$

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]

$$E_D^P \vdash ((\omega_1, \nu_1, \phi_1), ..., (\texttt{BCAST(x)};\omega_i, \nu_i, \phi_i), ..., (\omega_n, \nu_n, \phi_n)) \xrightarrow{\tau} ((\omega_1, \nu_1, \phi_1'), ..., (\omega_i, \nu_i, \phi_i'), ..., (\omega_n, \nu_n, \phi_n'))$$

$$\text{where } \forall j \in [1..n] : \phi_j' = \phi_j \cdot (x, \nu_i(x), i)$$

The other rules described below are local rules; i.e. defining how a local process makes a move. We use the interleaving rule to make these moves global. 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. Therefore, this rule dictates the cyclic behavior of each process.

[Cycle start]

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

$$\text{Where } V_i^i = Var^{in}(P) \cap V_i,\ V_i^o = Var^{out}(P) \cap V_i$$

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 broadcasted to all other processes. Moreover, the transition is labelled 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. Remember 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. In $_{\mathrm{d}}\mathrm{SL}_\Diamond$, the first message from any sender may be received by the process. We use the source field of a message to determine which is the first message sent by a given process.

[Message treatment]

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

$\phi_i = (x_1, v_1, s_1) \cdots (x_{k-1}, v_{k-1}, s_{k-1}) \cdot (x_k, v_k, s_k) \cdot \phi''$
$\phi_i' = (x_1, v_1, s_1) \cdots (x_{k-1}, v_{k-1}, s_{k-1}) \cdot \phi''$
$\forall l \in [1..k-1] : s_l \neq s_k$. This is the first message sent by k. Where $W_{i/\tilde{x}} = \{w \in W_i \mid \tilde{x} \in Var(Cond(w))\}$

[End of message treatment]

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

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. On the other hand, if the FIFO channel is empty, the end of message treatment rule is the only one that may be applied.

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 broadcasted and the WHENs that may be triggered by this assignment need to be scheduled for treatment.

[Assignment]

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

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

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.

[If]

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

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

[Output]

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

Using those semantics rules, given a well-formed $_{\mathrm{d}}\mathrm{SL}_\Diamond$ program P and a distribution D of P, we can define a labelled transition system describing the behavior of P, in D.

Definition 2.12 - Distributed semantics of a $_{\mathrm{d}}\mathrm{SL}_\Diamond$ program

Given a well-formed $_{\mathrm{d}}\mathrm{SL}_\Diamond$ 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 a labelled transition system $(\mathcal{G}_D^P, G_D^0, (V(P) \times \{!, ?\} \times \{\top, \bot\}), \rightarrow)$ where:

- $G_D^0 = ((\omega_1, \nu_1, \phi_1), ..., (\omega_n, \nu_n, \phi_n))$ where $\forall i \in [1..n]$,
  - $\omega_i = \varepsilon$
  - $\nu_i(x) = \bot$, $\forall x \in (V_i \cup \widetilde{Var(P)} \cup OldCond(W_i))$
  - $\phi_i = \varepsilon$
- $\rightarrow$ 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 semantics rule given previously.

Chapter 3

Analysis of distributions

In the previous chapter, we described the $_{\mathrm{d}}$SL's syntax and semantics. We saw that a $_{\mathrm{d}}$SL program may be run using different distributions, and the semantics of a $_{\mathrm{d}}$SL program P was dependent on the distribution used. In this chapter, we show a relationship between different distributions of a $_{\mathrm{d}}$SL program. This result was first presented in [WGMM04]. Using the distribution hierarchy of definition 2.4, we can define a weak *simulation* relation between different semantics of P (i.e. using different distributions). This simulation relation allows us to build a *lattice* between the semantics using different distributions and this lattice is useful for verification. We first recall the notion of *simulation* relation, then we prove that we have a weak simulation between semantics using different distributions of a $_{\mathrm{d}}$SL program P. In this chapter, we will use the semantics defined in the previous chapter, thus we consider only $_{\mathrm{d}}\mathrm{SL}_{\Diamond}$ programs. Full-featured $_{\mathrm{d}}$SL programs can be translated in $_{\mathrm{d}}\mathrm{SL}_{\Diamond}$ using the transformation described in section 2.5.

3.1 Simulation relation

In the previous chapter, we defined a labelled transition system. This structure is used to describe the behavior of a $_{\mathrm{d}}\mathrm{SL}_{\Diamond}$ program using a given distribution. We now define a way to compare the different LTS's generated from the same $_{\mathrm{d}}\mathrm{SL}_{\Diamond}$ program using different distributions. We first define the notion of simulation relation.

Definition 3.1 - Weak simulation relation

Given two labelled 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

$$\widehat{a} \in \begin{cases} \tau^* \cdot a \cdot \tau^* & \text{if } a \in \Sigma \\ \tau^* & \text{if } a = \tau \end{cases}$$

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 simulation relation $\mathcal{R}$ such that $(q_1^0, q_2^0) \in \mathcal{R}$.

In the previous chapter, we have defined the behavior of a program P with distribution D as a labelled 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 behavior. For that, we need some intermediate results. Indeed, we first prove there is a simulation relation between two distributions having the one-split property defined below.

Definition 3.2 - One-split property
Let P be a $_{\mathsf{d}}\mathsf{SL}_{\Diamond}$ program, and D, D' be two distributions of P. We say that D' is a one-split of D if and only if :

- $D = \{V_1, \ldots, V_n\} \wedge D' = \{V'_1, \ldots, V'_{n+1}\}$
- $V_i = V'_i,\ \forall i \in [1, n-1]$.
- $V_n = V'_n \cup V'_{n+1}$

Before proving this result, we first give two definitions and a lemma about distributions. These will be useful during the proof of simulation.

Definition 3.3 - Workload distribution
Let $\omega, \omega_1, \omega_2$ be three workloads (i.e. sequences of instructions, including extended instructions). We have that (ω_1, ω_2) is a distribution of ω, noted $(\omega_1, \omega_2) \succ \omega$, if and only if one of the following holds :

$$
\begin{array}{l}
\omega = \epsilon \wedge \omega_1 = \epsilon \wedge \omega_2 = \epsilon \\
\omega = x; \omega' \text{ and one of the following holds:} \\
\quad x = \mathtt{MSG} \wedge \omega_1 = \mathtt{MSG};\omega'_1 \wedge \omega_2 = \mathtt{MSG};\omega'_2 \wedge (\omega'_1, \omega'_2) \succ \omega' \\
\quad x \notin \{\mathtt{MSG}, \epsilon\} \wedge \omega_1 = x; \omega'_1 \wedge (\omega'_1, \omega_2) \succ \omega' \\
\quad x \notin \{\mathtt{MSG}, \epsilon\} \wedge \omega_2 = x; \omega'_2 \wedge (\omega_1, \omega'_2) \succ \omega'
\end{array}
$$

When we have $(\omega_1, \omega_2) \succ \omega$, we will show that, with the correct interleaving, the execution of ω_1 in parallel with ω_2 will have the same semantics than the execution of ω, provided that the same context is used.

Definition 3.4 - FIFO distribution
Let ϕ be a FIFO with n possible senders, ϕ' be a FIFO with $n+1$ possible senders. We have that ϕ' is a distributed version of ϕ, noted $\phi' \approx_n \phi$, if and only if one of the following holds:

$$
\begin{array}{l}
\phi = \epsilon \wedge \phi' = \epsilon \\
\phi = (x, v, s) \cdot \phi_n \wedge \phi' = (x, v, s') \cdot \phi'_n \wedge \phi'_n \approx_n \phi_n \\
\text{with} \\
\quad s \neq n \rightarrow s' = s \\
\quad s = n \rightarrow (s' = n \vee s' = n + 1)
\end{array}
$$

When we have $\phi' \approx_n \phi$, we know that ϕ' has almost the same messages than ϕ. The only difference is for the messages coming from the "splitted" site. In this case, the source needs to be updated to the new owner of the corresponding variable.

Lemma 3.1
Given a well-formed $_{\mathsf{d}}$SL program P and $D = (V_1, \cdots, V_n)$, $D' = (V'_1, \ldots, V'_n, V'_{n+1})$ two distributions of P 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 :

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

This proof is immediate. Now we can give an important lemma stating that we have a simulation relation between two semantics for distribution having the one-split property.

Lemma 3.2 - One-split simulation
Given a well-formed $_{\mathsf{d}}$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
We define a relation $\mathcal{R} \subseteq \mathcal{G}^P_D \times \mathcal{G}^P_{D'}$ such that if $G = ((\omega^G_1, \nu^G_1, \phi^G_1), (\omega^G_2, \nu^G_2, \phi^G_2), ..., (\omega^G_n, \nu^G_n, \phi^G_n))$ and $G' = ((\omega^{G'}_1, \nu^{G'}_1, \phi^{G'}_1), (\omega^{G'}_2, \nu^{G'}_2, \phi^{G'}_2), ..., (\omega^{G'}_n, \nu^{G'}_n, \phi^{G'}_n), (\omega^{G'}_{n+1}, \nu^{G'}_{n+1}, \phi^{G'}_{n+1}))$, $(G, G') \in \mathcal{R}$ if and only if:

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

We prove that R is a weak 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 notations :

- $H = ((\omega^H_1, \nu^H_1, \phi^H_1), (\omega^H_2, \nu^H_2, \phi^H_2), ..., (\omega^H_n, \nu^H_n, \phi^H_n))$
- $H' = ((\omega^{H'}_1, \nu^{H'}_1, \phi^{H'}_1), (\omega^{H'}_2, \nu^{H'}_2, \phi^{H'}_2), ..., (\omega^{H'}_n, \nu^{H'}_n, \phi^{H'}_n), (\omega^{H'}_{n+1}, \nu^{H'}_{n+1}, \phi^{H'}_{n+1}))$.

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 one of the two global semantics rules: interleaving or broadcast.

[Broadcast] According to the broadcast semantics 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 $(\omega_i^G = \omega_i^{G'} \wedge \nu_i^G = \nu_i^{G'} \wedge \phi_i^{G'} \approx_n \phi_i^G)$. 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 same message is added to all FIFO channels in H and in H'. Therefore, FIFO channels in H and H' respect the relation $\approx_n$. 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 semantics 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 in H and H' respect the relation $\approx_n$. It is then easy to see that $(H, H') \in \mathcal{R}$.

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

[Interleaving] According to the interleaving semantics 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 $(\omega_i^G = \omega_i^{G'} \wedge \nu_i^G = \nu_i^{G'} \wedge \phi_i^{G'} \approx_n \phi_i^G)$. If this rule not the message treatment rule, then it can also be applied to $(G')_i$. It follows that:

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

   Only the local process i changes, the other local processes remain unchanged. Therefore, since $(\omega_i^G = \omega_i^{G'} \wedge \nu_i^G = \nu_i^{G'} \wedge \phi_i^{G'} \approx_n \phi_i^G)$, we have that $(H, H') \in \mathcal{R}$.

   If the message treatment rule is applied in $(G)_i$, then we must prove that it can also be applied in $(G')_i$ and that the resulting state $(H')_i$ is such that $(H, H') \in \mathcal{R}$. We have that $\omega_i^{G'} = \omega_i^G = MSG;\eta_i H$. If the message treatment rule could be applied, we have that $\phi_i^H = {\phi'}_i^H \cdot (x, v, s) \cdot {\phi''}_i^H$ and, as $\phi_i^{H'} \approx_n \phi_i^H$, the same message can be read from $(G')_i$.
   The case where $s \neq n$ is trivial, as the FIFO queues are identical for messages with sender $s \neq n$. If $s = n$, then, we still are able to read the message since (x, v, s) is the first message in ϕ_i^G with $s = n$ which makes (x, v, s') the first message in $\phi_i^{G'}$ with $s' \in \{n, n+1\}$. In the resulting state $(H')_i$, we have that $\omega_i^{H'} = \omega_i^H \wedge \nu_i^{H'} = \nu_i^H \wedge \phi_i^{H'} \approx_n \phi_i^H$. Thus, $(H, H') \in \mathcal{R}$.

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 $\omega_n^G = \epsilon$, $(Here, a = \tau)$. 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
$(G')_n \xrightarrow{\tau} (G'')_n \wedge \forall i \neq n \ : \ (G')_i = (G'')_i$ Cycle start in local process n
$(G'')_{n+1} \xrightarrow{\tau} (H')_{n+1} \wedge \forall i \neq n+1 \ : \ (G'')_i = (H')_i$ Cycle start in local process n+1

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

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

Indeed, the valuations on the variables and the FIFO channels remain the same when the cycle start rule is applied. Let's look at ω_n^H, by lemma 3.1, we know that every input/output/when treatment instructions 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 $(H, H') \in \mathcal{R}$.

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

(a) if $\omega_n^{G'} = \mathtt{INPUT}(x); \eta_n^{H'}$ and $\omega_{n+1}^{H'} = \omega_{n+1}^{G'}$, we can apply the input semantics 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 $\nu_n^{H'} = \nu_n^{H'}[x \mapsto v]$. 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 :

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

We can conclude that $(H, H') \in \mathcal{R}$.

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

$[\mathbf{Message\ treatment}]$ In this case, we have $\omega_n^G = MSG; \eta_n^H$, $(Here, a = \tau)$. 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, then we have $\phi_n^G = \phi' \cdot (x, v, s) \cdot \phi''$, as $\phi_n^{G'} \approx_n \phi_n^G$ and $\phi_{n+1}^{G'} \approx_n \phi_{n'}^G$, the same message can be read by the three processes. Indeed, the case where $s \neq n$ is trivial, as the FIFO queues are identical for messages with sender $s \neq n$, if $s = n$ then we still are able to read the message since (x, v, s) is the first message in ϕ_n^G with $s = n$ which makes (x, v, s') the first message in $\phi_n^{G'}$ and $\phi_{n+1}^{G'}$ with $s' \in \{n, n+1\}$. 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 3.1, 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 $(H, H') \in \mathcal{R}$.
In all the cases, we have $G' \xrightarrow{\tau \cdot \tau} H'$ and $(H, H') \in \mathcal{R}$.

$[\textbf{Assignment}]$ In this case, we have $\omega_n^G = \mathtt{x} := \mathtt{e}; \eta_n^H, (Here, a = \tau)$. 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 $\nu_{n+1}^{H'}$, $\nu_{n+1}^{H'}$ 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 $(\eta_n^{H'}, \omega_{n+1}^{G'}) \succ \omega_n^{H'}$, it is straightforward to see 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 all the cases ($x \in V(p)$ or $x \in OldCond(P)$), we have $G' \xrightarrow{\tau} H'$ and $(H, H') \in \mathcal{R}$.

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

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

(a) if $\omega_n^{G'} = \texttt{IF } e \texttt{ THEN } \omega^\top \texttt{ ELSE } \omega^\perp \texttt{ ENDIF}; \eta_n^{H'}$ 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 afterwards. We then have :

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

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

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

[Output] In this case, we have $\omega_n^G = \texttt{OUTPUT}(x); \omega_n^H$, $(Here, a = x!v)$. 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 $(H, H') \in \mathcal{R}$.

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

Finally, it easy to prove that $(G_D^0, G_{D'}^0) \in \mathcal{R}$. Indeed, according to definition 2.12, both in G_D^0 and $G_{D'}^0$, the workloads are empty, the valuations are assigning all variables to $\perp$ and the FIFO channel are empty. Therefore, by definition of $\mathcal{R}$ and by definition 3.3, we have $(G_D^0, G_{D'}^0) \in \mathcal{R}$ and :

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

■

Lemma 3.2 proves that, having two distributions D and D' such that the last set of variables of D is split in two in D', $[\![P]\!]_D \lesssim [\![P]\!]_{D'}$. We now prove that for any distributions D, D' such that $D \preceq D'$, $[\![P]\!]_D \lesssim [\![P]\!]_{D'}$. For this, we first give a lemma that will allow us to relate this problem to lemma 3.2.

Lemma 3.3 - 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 a one split of D_{i-1} ■

With this lemma, we know that if $D \preceq D'$, then we can construct a sequence of distributions and we can apply lemma 3.2 between each consecutive pair of distributions of this sequence.

Theorem 3.1 - Simulation

Given a well-formed $_{\mathsf{d}}\mathsf{SL}_\Diamond$ 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 $[\![P]\!]_D$:

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

Proof

This is a direct consequence of lemma 3.2, lemma 3.3, and the transitivity of the simulation relation. ■

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

Corollary 3.1 - Distribution lattice

Given a well-formed $_{\text{d}}\text{SL}_{\Diamond}$ program. Let $D_{min} = \{Var(P)\}$, D_{max} be the maximal distribution. 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.■

This lattice may be helpful for verification of $_{\text{d}}\text{SL}_{\Diamond}$ (and, thus, $_{\text{d}}\text{SL}$). Indeed, when a safety property is verified for a given distribution D, it is verified for all distribution $D' \preceq D$. On the other hand, when a counter-example is found, for a given distribution D, we know we have a counter-example for all distributions D' such that $D \preceq D'$. Note that De Wachter, Massart and Meuter already performed some experiments on the verification of $_{\text{d}}\text{SL}$ using different distributions in [DMM03].

This chapter presented an interesting theoretical result regarding distributions in $_{\text{d}}\text{SL}$. The next chapter presents a technique that may also be helpful to perform the verification of $_{\text{d}}\text{SL}$ programs.

Chapter 4

Slicing

In this chapter, we detail how to perform slicing on $_{\mathrm{d}}$SL. The purpose of slicing is to derive a subprogram containing only the instructions we are interested in to verify some property. Given a property φ we want to check on a $_{\mathrm{d}}$SL program P, we can compute a new program P' containing only the instructions needed to verify φ. This avoids useless computations during model-checking. We first start by giving the basic definitions commonly used in slicing. We use the formalism of *dependence graph* presented in [OO84]. Then, we explain how to adapt this formalism to $_{\mathrm{d}}$SL and we go through the numerous problems we faced during the implementation of a slicer for $_{\mathrm{d}}$SL. Indeed, $_{\mathrm{d}}$SL is an event-driven language, and we don't have a clear control flow. Moreover, the distributed, asynchronous features may introduce some new dependencies.

Section 4.1 presents works already performed on slicing. In section 4.2, we present a slicing formalism for $_{\mathrm{d}}$SL. Then, in section 4.3, we explain how to use this formalism to perform slicing. When trying to apply the usual technique to $_{\mathrm{d}}$SL, we encountered some problems leading to over-conservative slices. Section 4.3 and section 4.4 discuss these problems. Finally, an experimental evaluation of an implementation of the slicer is presented in section 4.5. In this chapter, some new notations are introduced. A summary of the most important ones can be found in appendix C.

4.1 Related Works

Slicing was first introduced by Weiser in [Wei81]. In his approach, slices were computed by considering sets of relevant instructions. Using what he called *dataflow equations*, he computed a fixpoint, starting from the slicing criterion (i.e. the instructions he was interested in), to obtain a slice. Weiser also showed that computing a minimal slice (i.e. there is no other slice smaller than this one for the same criterion), was undecidable. Inter procedural slicing was also studied in this paper.

In [BC85], Bergeretti and Carr defined the *information flow* approach. This approach uses information-flow relations. These relations describe the link between different instructions using the defined variables, used variables, and the type of instructions. These relations were also defined for list of instructions and, by applying it to the whole program, starting from the slicing criterion, they could compute a slice. This paper also studies inter procedural slicing.

In [OO84], Ottenstein used the *program dependence graph* formalism, introduced in [KKP+81]. Here, the dependences between instructions are modeled as graph edges. And slicing is performed using backward reachability algorithms. The most important part is computing the program dependence graph. This technique is refined for inter procedural slicing by Horwitz *et al.* in [HRB90]. For this, they introduce the *system dependence graph* where inter procedural edges are considered. A survey of the work that has been done by Horwitz, Reps *et al.* on slicing and other program analyse can be found in [HR92].

A more complete survey of slicing techniques can be found in [Tip95].

4.2 The dependence graph

To perform slicing, we will use a formalism based on the program dependence graph (PDG) of [OO84]. Informally, a PDG is a directed graph where instructions are represented by vertices and an edge between two vertices represents a dependence between these vertices. Using the formalism of PDG, slicing can be reduced to a simple backward reachability procedure starting from an initial set of vertices, called the slicing criterion. In this section, we give a formal definition of a $_{\mathrm{d}}$SL *statement level* control flow graph which is a formalism used to represent the control flow of a $_{\mathrm{d}}$SL program. This definition is used to define the $_{\mathrm{d}}$SL system dependence graph, which is a derivate of the PDG based inter procedural slicing techniques described in [HRB90].

Definition 4.1 - $_{\mathrm{d}}$SL statement level control flow graph (CFG for short).
Let P be a well-formed $_{\mathrm{d}}$SL program, we define its control flow graph as a directed graph $C = (V_C, E_C)$ where :

- $V_C = V_a \cup V_{ep} \cup V_{call} \cup V_{if} \cup V_{while}$ is the set of nodes of the CFG where :
  - V_a is the set of assignements of P.
  - V_{ep} is the set of entry and exit points of P. This set can be partitioned in two ways. We have that $V_{ep} = V_\top \cup V_\bot$, i.e. $V_\top$ is the set of entry points of P and $V_\bot$ is the set of exit points of P. In the following, for an entry e, we will note e its entry point in C and $\bot_e$ its exit point.

    Furthermore, $V_{ep} = V_{meth} \cup V_{seq} \cup V_{when}$. V_{meth} (resp. V_{seq}, V_{when}) is the set of vertices (entry and exit) for the `METHOD`s (resp. `SEQUENCE`s, `WHEN`s) of P.
  - $V_{call} = V_{acall} \cup V_{scall}$ is the set of calls of P. V_{acall} is the set of asynchronous calls (e.g. calls of the form : `LAUNCH M(P1,P2);`) and V_{scall} is the set of synchronous calls (e.g. calls of the form : `M(P1,P2);`).
  - V_{if} is the set of `IF` instructions of P.
  - V_{while} is the set of `WHILE` instructions of P.
- E_C is the set of edges of the CFG. $(v, v') \in E_C$ if the control may flow from v to v'.

We note $\rho(C)$ the set of finite paths $\rho \equiv v_1 \to v_2 \to \ldots \to v_n$ of C if $(v_i, v_{i+1}) \in E_C,\ \forall i \in [1, n-1]$.

For a given $_{\mathrm{d}}$SL program P, we note CFG(P) the control flow graph of P. CFG(P) gives the structure of the various entries of the program P (i.e. `WHEN`'s, `SEQUENCE`s and `METHOD`s) We define the following functions on the CFG :

- ENTRYOF : $V_C \to V_{ep}$. ENTRYOF (v) returns the entry point containing v.
- PARAM : $V_{ep} \to 2^{Var_A(P)}$. PARAM (e) gives the set of formal parameters of entry point e.

- PARAM : $V_{call} \times Var(P) \rightarrow expression$. PARAM (c, x) returns the expression used to initialise formal parameter x in call c.
- CALLED : $V_{call} \rightarrow V_{ep}$. CALLED (c) returns the entry point actually called by c.
- DEF : $V_C \rightarrow 2^{Var_A(P)}$ that gives the set of variables modified by the corresponding instruction.
- USE : $V_C \rightarrow 2^{Var_A(P)}$ that gives the set of variables used in an expression of the corresponding instruction
- USE $_\sim$: $V_C \rightarrow 2^{Var(P)}$ that gives the set of variables used in a tilded way.

Note that the functions DEF, USE, and USE $_\sim$will be extended later for the vertices of the dependence graph. Note that the functions used in chapter 2 (and defined in section 2.3) are still used in this chapter.

4.2.1 The system dependence graph.

This section gives the details of the construction of a SDG $G = (V, E)$ from a CFG $C = (V_C, E_C)$ of a $_{\text{d}}$SL program P. This will be done by analysing the CFG and finding some dependences between instructions. Weiser showed in [Wei81] that the problem of finding a *precise* slice is undecidable for abitrary programs. Thus, we can only compute an over-conservative slice. A formal definition of the system dependence graph is given below.

Definition 4.2 - $_{\text{d}}$SL System Dependence Graph (SDG for short)
Let P, be a well-formed $_{\text{d}}$SL program, C its CFG. The $_{\text{d}}$SL system dependence graph of C is a directed graph $G = (V, E)$ where :

- $V = V_{\mathsf{n}} \cup V_{\mathsf{ai}} \cup V_{\mathsf{ao}} \cup V_{\mathsf{fi}} \cup V_{\mathsf{fo}}$. More formal definitions of these vertices are given later in this section.
  - V_{n} is the usual set of vertices (i.e. corresponding to the vertices of the control flow graph).
  - V_{ao} is the set of `actual out` vertices.
  - V_{ai} is the set of `actual in` vertices.
  - V_{fo} is the set of `formal out` vertices.
  - V_{fi} is the set of `formal in` vertices

  The last four types of vertices are used to describe dependences due to parameter passing during calls and triggering and will sometimes be called *fake vertices* in this document.

- $E = E_{\mathsf{f}} \cup E_{\mathsf{call}} \cup E_{\mathsf{ct}} \cup E_{\mathsf{t}} \cup E_{\mathsf{par}}$. More formal definitions of these edges are given later in this section.
  - E_{f} is the set of flow edges. A flow edge describes a dependence that occurs due to the modification and usage of a given variable by two different instructions. Furthermore E_{f} can be divided into E_{fs} and E_{fa}. E_{fs} is the set of synchronous flow dependence and E_{fa} is the set of asynchronous flow dependence (i.e. through tilded variables and `LAUNCH`'s).
  - E_{call} is the set of call edges. $(u, v) \in E_{\mathsf{call}}$ if u is a call vertex and v is an entry vertex in C. Again, E_{call} will be divided in E_{ca} and E_{cs}, respectively the asynchronous calls and the synchronous calls.

- E_{ct} is the set of control edges. A control edge describes a dependence that occurs between a controlling instruction (i.e. an `if`, a `while` or an entry point) and a controlled instruction.
- E_{t} is the set of trigger edges. $(u, v) \in E_{\mathsf{t}}$ iff v is a `WHEN` entry vertex and u is an assignment that may trigger v. E_{t} can be divided into E_{ts} and E_{ta}, respectively the synchronous and asynchronous triggers. Asynchronous triggers come from use of tilded variables in the conditions of `WHEN`'s.
- E_{par} is the set of edges inserted to describe the parameter passing during triggerings and calls. E_{par} can be divided into E_{pi}, the edges corresponding to copy-in of the parameters and E_{po}, the edges corresponding to the copy-out of the parameters. Indeed, as explained later, parameter passing will be modeled by a copy-in/copy-out mechanism.

Having a dependence graph G, we may construct a subgraph G' by tearing away some vertices. From this new graph, we may sometimes deduce a new $_{\mathrm{d}}$SL program. The slicing will consist in computing, from a dependence graph G of a program P, a subgraph G' such that the $_{\mathrm{d}}$SL program P' built from P and G' has the same semantic behavior as P w.r.t. a slicing criterion.

Before giving more details on the vertices and edges of the SDG, we first explain how to handle two features of $_{\mathrm{d}}$SL: global variables and `WHEN`'s. As in [HRB90], global variables are handled as in-out parameters. Indeed, global variables are copied-in before the call to a procedure and are copied-out after the procedure as in/out parameters. `WHEN`'s are treated as a simple procedure, everytime an assignment may trigger a given `WHEN`, we consider this assignment as a fake call (whose parameters are only the global variables). From now on, we will use the following function :

- NORM : $V \rightarrow V_C$. Intuitively, this function tells us where would be located a given vertex in the control flow. Using this function, we are able to precisely locate the DEF/USE of a vertex. Indeed, during the computation of the system dependence graph, fake vertices will be inserted, and NORM will be used to locate these vertices in the control flow. This localisation will be used to compute dependencies (i.e. edges). A more formal definition of NORM will be given when the set V of vertices is detailed.

We now detail the set V of vertices.

Vertices in the SDG.

We now define precisely the set V of vertices. We start with the usual vertices. These vertices are simply a copy of the vertices of the original CFG.

Usual vertices

$V_{\mathrm{n}} \equiv V_C$.
Moreover, let $\textsc{norm}(v) = v,\ \forall v \in V_{\mathrm{n}}$.

We now detail the creation of special vertices used for calls and triggerings. These vertices will be used to perform *inter procedural slicing* as described in [HRB90]. In [HRB90], fake assignements are introduced to simulate the parameter passing. Indeed, everytime we meet a call, fake assignements are inserted to initialise the in-parameters at the call side. Furthermore, fake assignements are also inserted to copy the value of the out-parameters at the end of the procedure. These fake assignments correspond to the vertices in $V_{\mathrm{ai}} \cup V_{\mathrm{ao}}$. Then, control edges are inserted from the call instruction to these fake assignments.

On the called side (i.e. the entry point), fake assignments are inserted at the beginning of the procedure to copy-in in-parameters, the corresponding vertices are vertices from the set V_{fi}. Fake assignments are also inserted at the end of the procedure to copy-out out-parameters, the corresponding vertices are vertices from the set V_{fo}. Then, control edges are inserted from the entry (for V_{fi}) or exit (V_{fo}) point to these assignments.

Figure 4.1 gives an example of calls are modeled using `actual in/out` and `formal in/out` vertices.

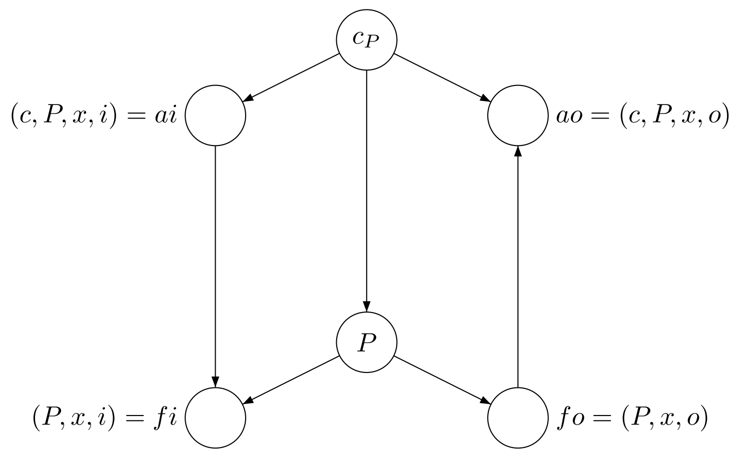

Figure 4.1: How calls are modeled in the SDG

First, we describe the set of `actual in` vertices $V_{\text{ai}} = V_{\text{ci}} \cup V_{\text{trig\_i}} \cup V_{\text{param}}$ and the set of `actual out` vertices $V_{\text{ao}} = V_{\text{co}} \cup V_{\text{trig\_o}}$. The set V_{param} will be described later, when considering parameters passing.

Handling calls

- $V_{\text{ci}} = \{(c, e, x, i) \mid c \in V_{scall} \wedge e = \text{CALLED}(c) \wedge x \in Var(P)\}$.
- $V_{\text{co}} = \{(c, e, x, o) \mid c \in V_{scall} \wedge e = \text{CALLED}(c) \wedge x \in Var(P)\}$.

where i (resp. o) stands for *input* (resp. *output*). For each synchronous call and for each variable, we associate two vertices in V that will be used to express dependences due to global variables when entries are called. Moreover, we pose $\text{DEF}\ ((c, e, x, o)) = \text{USE}\ ((c, e, x, i)) = \{x\}$ and $\text{NORM}((c, e, x, o)) = \text{NORM}((c, e, x, i)) = c$. Note that these vertices are only inserted for synchronous calls. Asynchronous calls are considered in section 4.2.5.

`WHEN` triggering

- $V_{\text{trig\_i}} = \{(a, w, x, i) \mid a \in V_a \wedge w \in V_{when} \wedge x \in Var(P) \wedge \text{DEF}(a) \cap \text{USE}(\text{COND}(w)) \neq \emptyset\}$.
- $V_{\text{trig\_o}} = \{(a, w, x, o) \mid a \in V_a \wedge w \in V_{when} \wedge x \in Var(P) \wedge \text{DEF}(a) \cap \text{USE}(\text{COND}(w)) \neq \emptyset\}$.

As explained below, synchronous triggering of `WHEN`'s will be treated as calls. Thus, for all assignment that may trigger a `WHEN`, we must insert vertices in V_{ai} and V_{ao} to simulate a call. Moreover, we pose USE $((a,w,x,i))$ = DEF $((a,w,x,o)) = \{x\}$ and NORM$((a,w,x,o))$ = NORM$((a,w,x,i)) = a$. These vertices are only inserted for synchronous trigger, asynchronous triggers are considered in section 4.2.5.

Now, we describe the `formal in` set of vertices and the `formal out` set of vertices. We have that $V_{\text{fi}} = V_{\text{inglobal}} \cup V_{\text{inparam}}$.

Entry points

- $V_{\text{inglobal}} = \{(e,x,i) | e \in V_{ep} \wedge x \in Var(\text{P})\}$
- $V_{\text{fo}} = \{(e,x,o) | e \in V_{ep} \wedge x \in Var(\text{P})\}$. Here, V_{fo}, only contains fake vertices for the global variables (remember parameters are always passed by value in $_{\text{d}}$SL).

These vertices will be used for the copy-in (before the execution of entry point e) and copy-out (after the execution of exit point e) of global variables. Moreover, we pose DEF $((e,x,i))$ = USE $((e,x,o)) = \{x\}$, NORM$((e,x,o)) = \perp_e$ and NORM$((e,x,i)) = e$.

We now explain how to handle parameters. In $_{\text{d}}$SL, parameters are passed by value, and we only have to perform a copy-in. This is represented by the set V_{inparam} on the called side and by the set V_{param} on the calling side.

Parameter passing

$V_{\text{inparam}} = \{(e,x,i) | e \in V_{ep} \wedge x \in \text{PARAM}(e)\}$.
These vertices will be used for the parameter passing. We define DEF$((e,x,i)) = \{x\}$, USE$((e,x,i)) = \emptyset$, and NORM$((e,x,i)) = e$.

Parameter passing on calling side

$V_{\text{param}} = \{(c,e,x,i) | c \in V_{call} \wedge x \in \text{PARAM}(\text{CALLED}(c)) \wedge e \in V_{ep}\}$.
This corresponds to the parameter passing on the calling side. Moreover, we pose USE$((c,e,x,i))$ = USE(PARAM(c,e,x,i)) (i.e. we are using the variable of the expression passed in parameter), DEF$((c,e,x,i)) = \emptyset$, and NORM$((c,e,x,i)) = c$.

4.2.2 Intra procedural edges in the SDG

We now detail the different sets of edges used to build the system dependence graph. Using these edges, a simple backward reachability algorithm may be applied to slice a $_{\text{d}}$SL program w.r.t. a given slicing criterion. Thus, to have a precise slice, one must be careful during the definition of dependence edges between vertices.

In the original intra procedural slicing technique [OO84], we had two main types of dependences (i.e. edges) in the program dependence graph, namely :

- Control dependences which model the fact that the execution of an instruction depends on another instruction.
- Data dependences which model the fact that the value of a variable at a given instruction may depend on another instruction.

Formal definitions of control and data dependences can be found in [Tip95].

Here, we must refine dependences classifications to take into account synchronous and asynchronous dependences, and inter procedural dependences. Next paragraphs detail the set of edges in the SDG, for *intra procedural* dependences. *Inter procedural* dependences are explained in section 4.2.3.

Control dependences Control dependences describe dependences between a vertex of the CFG and the last instruction that may decide of its execution, or the entry point of the procedure containing this instruction. For example, in the code of figure 4.2, the assignment at line 3 is control dependent on the `if` at line 2, which itself is control dependent on the entry point of line 1. The assignment at line 5 is control dependent on the entry point of line 1.

```
1: sequence P()
2:   if(x==4) then
3:      y := 3;
4:   end_if
5:   x := y;
6: end_sequence
```

Figure 4.2: Example of control dependence

To give a more formal definition of control dependence, we must define the notion of *post-domination*.

Definition 4.3 - post-domination
Let v, v' be two vertices of a CFG $C = (V_C, E_C)$, and let $e = \text{ENTRYOF}(v)$ be the entry point of v. We say that v' post-dominates v iff :

- $v \neq e$, i.e. v is not the entry point e.
- $\forall \rho \equiv v = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_n = \perp_e \in \rho(C),\ \exists i \in [0, n],\ v_i = v'$. All path in C from v to $\perp_e$ pass through v'.

For example, the assignment at line 5 post dominates the `if` at line 2 and the assignment at line 3. Now, we can define what is a control edge between two vertices in V. Let $C = (V_C, E_C)$ be the control flow graph of P, we define the control edge in E_{ct} as follow :

Definition 4.4 - Control edge - E_{ct}
Let $v, v' \in V_{\mathsf{n}}$ be two vertices of G. We have that $(v, v') \in E_{\mathsf{ct}}$ (i.e. v' is control dependent on v) if there exists a path $\rho \equiv v = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_n = v'$ in $\rho(C)$ from v to v' such that :

- $\forall i \in [1, n] v'$ post-dominates v_i.
- v' does not post-dominate v. Indeed, v can decide of the execution of v'.

These conditions ensure that v is the last instruction that may decide of the execution of v' or v is an entry point containing v'.

Furthermore, by convention, for all $v \in V \setminus V_{\mathsf{n}}$, we pose $(\text{NORM}(v), v) \in E_{\mathsf{ct}}$. Indeed, fake vertices are control dependent on the vertex that yielded to the creation of these fake vertices (i.e. the call, entry point or exit point).

A given vertex is control-dependent of the last vertex that may decide of its execution. This vertex may be an `if`, a `while`, or an entry point. Control edges are needed to keep the structure of the $_{\text{d}}$SL program after the slicing. Indeed, to be sure that we have the same behavior before and after the slicing, when an instruction is included is the slice, its controlling instruction (and its entry point) must also be included in the slice. In the example of figure 4.2, control edges would be inserted from the vertex for entry point of line 1 to vertices for line 2 and 5, and a control edge would be inserted from vertex for `if` of line 2 to vertex for line 3.

Note that this is different from the edges in the CFG. Indeed, the latter gives the sequencing of instructions in the $_{\text{d}}$SL program.

Flow dependences Synchronous flow dependences are used to express intra procedural data dependences. The dependences corresponds to the use of a variable previously defined in the procedure (inter procedural dependences will be explained in the next section). Using DEF and USE, we may simply define the set of synchronous flow edges E_{fs}. Let $C = (V_C, E_C)$ be the control flow graph of P, we define the control edge in $E_{\text{fs}} = E_{\text{fsu}} \cup E_{\text{cf}}$ where E_{fsu} is the set of "usual" synchronous flow edges. And E_{cf} is the set of flow edges inserted for the treatment of call/triggers.

Definition 4.5 - Usual Synchronous Flow edges - E_{fsu}
Let $v, v' \in V$ be two vertices of G. We have that $(v, v') \in E_{\text{fsu}}$ iff

- $v \neq v' \wedge \big(\text{NORM}(v) = \text{NORM}(v') \Rightarrow (v \in V_{\text{n}} \cup V_{\text{fi}})\big)$
- $\exists x \in Var(P), \text{DEF}(v) \ni x \wedge \text{USE}(v') \ni x$.
- There exists a path $\rho \equiv \text{NORM}(v) = v_0 \to v_1 \to \ldots \to v_n = \text{NORM}(v') \ \in \rho(C)$ from $\text{NORM}(v)$ to $\text{NORM}(v')$ such that : $\forall i \in [1, n-1]\ \text{DEF}(v_i) \not\ni x \ \wedge\ (x \in Var_A(P) \Rightarrow v_i \notin V_{scall})$.

Thus, a vertex v' is flow dependent on a vertex v iff v defines a variable x used by v' and there is a path in the CFG with no new definition of x from $\text{NORM}(v)$ to $\text{NORM}(v')$. If x is a global variable, this path should not contain synchronous calls either. Indeed, for every call, we suppose we have a copy-out of all global variables. This copy-out is considered as a new definition of x. Note that we allow assignment that may synchronously trigger some `WHEN` to be in the path : as, during the slicing, we don't try to determine whether this assignment will trigger this `WHEN`, we always suppose there is a path where this `WHEN` is not triggered.

Definition 4.6 - Call Flow edges - E_{cf}
Let $ai \in V_{\text{ai}}, ao \in V_{\text{ao}} with \text{NORM}(ai) = \text{NORM}(ao)$ be two vertices of G.

1. We have that $(ai, ao) \in E_{\text{cf}}$ iff : $ai = (c, e, x, i) \wedge ao = (c, e, x, o)$. These edges are inserted to "skip" a given call (more details below).
2. Moreover, we have that $(ao, ai) \in E_{\text{cf}}$ iff $ai = (c, w, x, i) \wedge ao = (c, w', x, o)$ and
   - $w' <_W w$
   - $\not\exists w''\ :\ (c, w'', x, o) \in V_{\text{ao}} \wedge w' <_W w'' \wedge w'' <_W w$

   These edges are inserted to model the sequence of treatment of `WHEN`s (more details below).

We also have a flow dependence between the `actual in/out` vertices. Condition 1 of definition 4.6 is used to "skip" a call (i.e. we are not sure that the called entry always redefines the corresponding variable). Condition 2 of definition 4.6 is for the treatment of `WHEN`'s triggering. Indeed, when an assignement triggers multiple `WHEN`'s, they are all executed one after the other following the order $<_W$ of appearance in the program code (see section 2.4 for a more formal explanation of `WHEN`'s treatment). We consider this `WHEN` treatment as a series of calls. Thus, we must insert these dependences to ensure that a modification of a variable in a previous `WHEN` is taken into account when considering the current `WHEN`.

Note that, following the definition E_{fs}, we may have flow edges from `actual out` or `formal in` vertices to other vertices in the procedure. On the other hand, we may also have flow edges from other vertices to `actual in` or `formal out` vertices. These edges will be used for inter procedural slicing.

In the example of figure 4.2, we would have a synchronous flow edge from vertex for line 3 to vertex for line 5. Moreover, we would have a synchronous flow dependence from vertex for line 3 to `formal out` vertex (P, y, o) and from vertex for line 5 to `formal out` vertex (P, x, o)

Here, we described intra procedural dependences. The next section discusses inter procedural dependences.

4.2.3 Inter procedural edges

Until now, we have considered only local definitions for data dependences of a given instruction. This section explains how inter procedural dependences can be inserted by considering special vertices corresponding to parameter passing. This technique comes from [HRB90].

Trigger dependences Trigger dependences are used to describe the dependences between the definition of a variable and the `WHEN`'s that may be triggered by this modification of the valuation. We have two kinds of trigger edges, the synchronous and the asynchronous ones. As explained later, the fact that a synchronous trigger corresponds to an instantaneous execution of the target `WHEN` yields new possible inter procedural flow dependences. On the other hand, an asynchronous trigger leads to a delayed execution, and no synchronous flow edges should be inserted, that's why a distinction is made between synchronous and asynchronous trigger edges. Let $C = (V_C, E_C)$ be the control flow graph of P, we define the trigger edges in E_{t} as follow :

Definition 4.7 - Trigger edges - E_{t}
Let $v, w \in V_{\mathsf{n}}$ be two vertices of G, such that $w \in V_{when}$.

- We have that $(v, w) \in E_{\mathsf{ts}}$ iff $\text{DEF}(v) \cap \text{USE}(w) \neq \emptyset$.
- Moreover, we have that $(v, w) \in E_{\mathsf{ta}}$ iff $\text{DEF}(v) \cap \text{USE}_{\sim}(w) \neq \emptyset$.

$E_{\mathsf{t}} = E_{\mathsf{ts}} \cup E_{\mathsf{ta}}$.

Call dependences Call dependences can be considered as a derivative of control dependences between a call instruction and the called entry. As for trigger dependences, we must make the distinction between synchronous and asynchronous call of an entry. Let $C = (V_C, E_C)$ be the control flow graph of P, we define the control edges in E_{c} as follow :

Definition 4.8 - Call edges - E_{call}
Let $c, e \in V_{\text{n}}$ be two vertices of G, such that $c \in V_{call} \wedge e = \text{CALLED}(c)$.

- We have that $(c, e) \in E_{\text{cs}}$ iff $c \in V_{scall}$.
- Moreover, we have that $(c, e) \in E_{\text{ca}}$ iff $c \in V_{acall}$.

$E_{\text{call}} = E_{\text{cs}} \cup E_{\text{ca}}$.

We now define edges from the `actual in` to the `formal in` and from the `formal out` to the `actual out`. These edges will make the link between the procedures and the corresponding calls for each variable/parameter. Let $C = (V_C, E_C)$ be the control flow graph of P, we define the control edges in E_{pi} and E_{po} as follow :

Definition 4.9 - Parameter in/out edges - E_{par}
Let $fi = (e, x, i) \in V_{\text{fi}}$, where $e \in V_{ep}$ is an entry (`WHEN`, `METHOD`, or `SEQUENCE`). We have that :

- $(ai, fi) \in E_{\text{pi}}$ iff $ai = (c, e, x, i)$.
- Moreover, if we have $fo = (e, x, o) \in V_{\text{fo}}$, we have : $(fo, ao) \in E_{\text{po}}$ iff $ao = (c, e, x, o)$.

$E_{\text{par}} = E_{\text{pi}} \cup E_{\text{po}}$.

4.2.4 Flow dependences of tilded use of variables

Due to the cycling behavior of $_{\text{d}}$SL, a given definition of a variable may influence the result of an instruction using a tilded copy of that variable, even if there is no explicit path in the CFG between the definition and the use. Here, we must consider all possible definitions of v in the $_{\text{d}}$SL program. This is due to the fact that message reading is non-deterministic in $_{\text{d}}$SL and we have no way of precisely determining which tilded message was read before a given use is performed. Next definition describes this relation (noted $E_{\sim}$) between vertices. Let $C = (V_C, E_C)$ be the control flow graph of P.

Definition 4.10 - $E_{\sim}$
Let $v, v' \in V_{\text{n}}$ be two vertices of G. We have that $(v, v') \in E_{\sim}$ iff $\text{DEF}(v) \cap \text{USE}_{\sim}(v') \neq \emptyset$.

This relation will be used to define the set E_{fa} of asynchronous flow dependences.

4.2.5 Managing entries that can be asynchronously called/triggered

One kind of asynchronous flow dependence remains to be considered. Indeed, in $_{\text{d}}$SL, some entry points may be called/triggered asynchronously. In this case, the entry does not have a precise call context (i.e. valuation for the global variables). Thus, in this case we cannot use the simple parameter passing mechanism described above for the global variables. Furthermore, `SEQUENCE`'s execution may be interrupted in $_{\text{d}}$SL (i.e. because of a code migration), and in this case, the context of each instruction of a `SEQUENCE` is hard to determine precisely. Let's look at the semantics of $_{\text{d}}$SL. First, it performs some input; in this case, `WHEN`'s conditionned on an input variable may be triggered. Once all event-driven processing is done, we enter in the message treatment phase. Here, some `WHEN`'s conditionned on a tilded copy of a variable may be triggered.

Moreover, SEQUENCEs and METHODs asynchronously called can be run. We also have some SEQUENCE whose execution have been interrupted and that are now resumed. On the other hand, some SEQUENCEs may be interrupted because of a migration of some of its code. Thus, we have four types of entry points that can be asynchronously run and whose call context is not precise :

- WHEN's conditionned on input variables.
- WHEN's conditionned on tilded copies of variables.
- METHODs that are asynchronously called.
- SEQUENCEs. For this kind of entry point, we have that all usages inside the SEQUENCE are undefined. For example in the following code :

```
sequence not_atomic()
  x:=3;
  (* not modifying x *)
  y:=x;
  x:=9;
end_sequence
```

  we have that the value of x at assignment y:=x is undefined. Indeed, as the SEQUENCE execution may be interrupted, we may have that some other code in the $_{\text{d}}$SL program may modify x between both assignments. Furthermore, the assignment x:=3 could be used in another place of the program, even if there is a new assignment x:=9 later in this SEQUENCE.

  As we can see on this example, using global variables directly in a SEQUENCE's body can be dangerous. Indeed, this variable may be modified by another entry point if the SEQUENCE's execution is interrupted.

For all these entries, we cannot make precise hypotheses about the call context. Moreover, the $_{\text{d}}$SL message treatment is non-deterministic, and we don't have a precise criterion to determine whether a given assignment *will* trigger a WHEN. Thus, we cannot clearly unfold the processing and the message treatment phase. The strategy we present here is based on the formal out and formal in vertices. Indeed, once all intra procedural flow dependences are computed, the formal out vertices are linked by flow edges to the last definitions of the corresponding variable. Symmetrically, flow edges from formal in vertices describe usages of variables that may not have been defined previously in the entry. We can use these vertices to model the asynchronous dependencies by inserting flow edges from formal out vertices of any entry point to formal in vertices for entry points that can be asynchronously run. The next definition describes this relation (denoted E_{ac}) between vertices. Let $C = (V_C, E_C)$ be the CFG of P.

Definition 4.11 - $E_{\textsf{ac}}$

Let $v \in V_{\textsf{fo}}, v' \in V_{\textsf{fi}}$ be two vertices of G. We have that $(v, v') \in E_{\textsf{ac}}$ iff $v = (e, x, o) \;\wedge\; v' = (e', x, i)$ and one of the following holds :

- $e' \in Whens(P) \;\wedge\; Var(Cond(e')) \cap Var^{in}(P) \neq \emptyset$. e' is a WHEN conditionned on an input variable.
- $\exists (c, e') \in E_{\textsf{ca}} \cup E_{\textsf{ta}}$. I.e. e' can be asynchronously called/triggered.

For the `SEQUENCE`s, as they can be interrupted, we have no other choice than inserting an edge to all vertices using the variable defined at the `formal out` vertex of all entry points. The next definition describes this relation (noted E_{seq1}) between vertices. Let $C = (V_C, E_C)$ be the CFG of P.

Definition 4.12 - E_{seq1}
Let $v \in V_{\mathsf{fo}}, v' \in V_{\mathsf{n}} \cup V_{\mathsf{ai}}$ be two vertices of G such that $\mathrm{ENTRYOF}(v') \in V_{seq}$. We have that $(v, v') \in E_{\mathsf{seq1}}$ iff :

- $v = (e, x, o)$.
- $x \in \mathrm{USE}(v')$.

Moreover, each definition inside the code of a `SEQUENCE` may be used at the beginning of any entry point that may be asynchronously called/triggered. Next definition describes this relation (noted E_{seq2}) between vertices. Let $C = (V_C, E_C)$ be the CFG of P.

Definition 4.13 - E_{seq2}
Let $v \in V_{\mathsf{fi}}, v' \in V_{\mathsf{n}} \cup V_{\mathsf{ao}}$ be two vertices of G such that $\mathrm{ENTRYOF}(v') \in V_{seq}$. We have that $(v', v) \in E_{\mathsf{seq2}}$ iff :

- $v = (e, x, i)$.
- $(c, e) \in E_{\mathsf{ta}} \cup E_{\mathsf{ca}}$.
- $x \in \mathrm{DEF}(v')$.

Using the relations defined in section 4.2.4 and 4.2.5, we may define precisely the edges from the set E_{fa}. Let $C = (V_C, E_C)$ be the CFG from P.

Definition 4.14 - Asynchronous flow edges - E_{fa}
The set E_{fa} of asynchronous flow edges is defined as follows :

$$E_{\mathsf{fa}} \equiv E_{\sim} \cup E_{\mathsf{ac}} \cup E_{\mathsf{seq1}} \cup E_{\mathsf{seq2}}$$

4.3 Applying reachability

Now that we have a dependence graph, we may apply slicing. A first approach would be to apply a simple backward reachability algorithm on this graph. We start from the vertex (or the set of vertices) corresponding to the slicing criterion, and we include all vertices that can transitively reach a vertex of the slicing criterion using dependence edges computed earlier. While this technique works well in single procedure programs [OO84], it proves itself too over-conservative in the inter procedural case. Section 4.3.1 discuss the call context problem, and section 4.3.2 gives a solution to this problem. To use this solution, we will need to introduce a new kind of edges, the summary edges, described in section 4.3.3.

During our implementation of the algorithm of [RHSR94], we discovered that this algorithm for computing summary edges was over-conservative. Hence, we propose a more refined approach of summary edges in section 4.3.4. Furthermore, an algorithm to compute these new summary edges is presented.

4.3.1 Call context problem

Let's consider the simple $_{\text{d}}$SL program of figure 4.3, figure 4.4 is its dependence graph. From now on, edges of the dependence graphs showed in this documents will be labeled by the set to which they belong (label f is put on all edges $e \in E_{\text{f}}$, e.g.).

```
when x==4 then //w1     method P()          when x==3 //w2
   P(); //c1             x:=x+1; //a           P(); //c2
   print(x); //printx   end_method          end_when
end_when
```

Figure 4.3: Call context problem

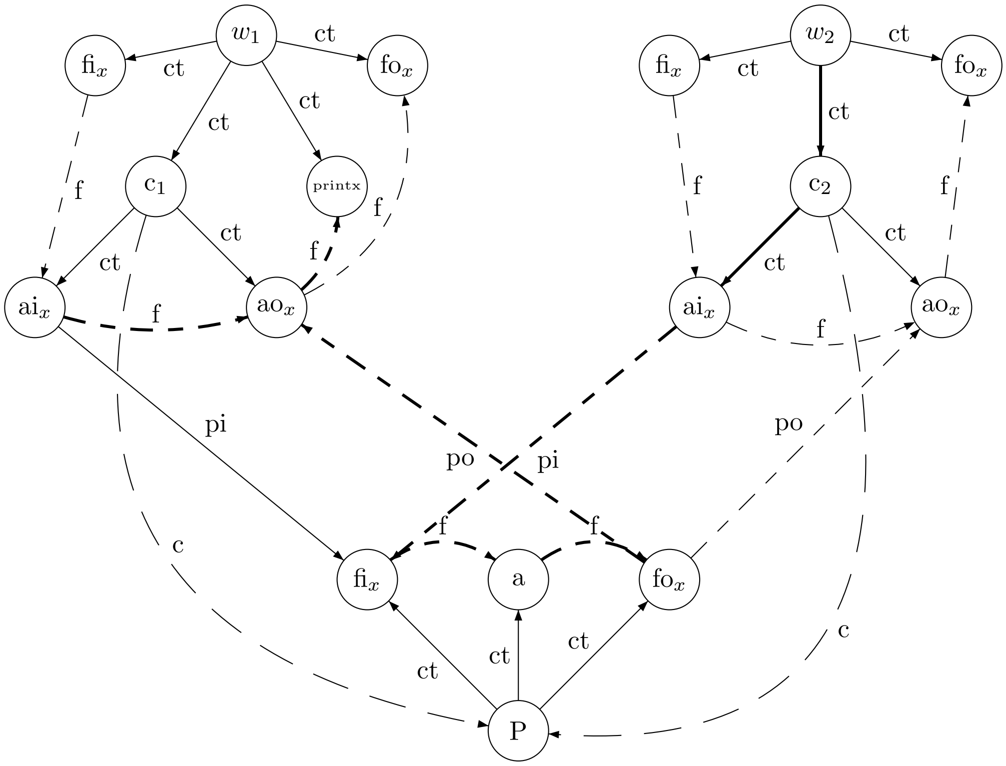

Figure 4.4: Dependence graph of figure 4.3 - bold edges represent a spurious path.

Now let's apply the backward reachability algorithm on the dependence graph with instruction `print(x)` as slicing criterion. We clearly see that procedure `P` and call `c1` to `P` must be included in the slice, so does the `when` entry point `w1`. But, if we follow bold edges of figure 4.4, we will also include the call `c2` and the corresponding `WHEN` in the slice. This is due to the fact that, when considering a given call, we may explore vertices of a different call by following other parameter edges. This is the call context problem. Intuitively, we can deduce that this kind of problem is caused by the exploration of non *realisable paths.* A realisable path is a path that respects the call context constraint, i.e. it always goes back to the call from where it comes from. To define realisable paths more formally, we number each call/trigger in a $_{\mathrm{d}}$SL program. Then, we label each edge in E_{pi} (resp. E_{po}) with the string "$(_i$" (resp. "$)_i$") where i is the number of the corresponding call/trigger. All other edges are labelled with "ϵ". We can say that a path is realisable if the string generated by the label on the edges belongs to rule 2 of the following grammar ([HRB90]) :

$$\begin{array}{llll} [1] & \mathit{matched} & \longrightarrow & (_i\ \mathit{matched}\ )_i\ \mathit{matched} \qquad \forall i \\ & & & \mid \epsilon \\ [2] & \mathit{realisable} & \longrightarrow & \mathit{realisable}\ (_i\ \mathit{matched} \qquad \forall i \\ & & & \mid \mathit{matched} \end{array}$$

4.3.2 Two phases slicing

In the previous section, we presented the call context problem that may arise if we apply a simple reachability to compute a slice. This section presents a possible solution to the call context problem. The idea, that comes from [HRB90], is to perform a backward exploration in two phases on the dependence graph. In each of these phases, some set of edges will not be followed to avoid the call context problem. The first phase, that starts from the slicing criterion, only "goes up" in the dependence graph (i.e., exploring only calling procedures). In practice, everytime a new vertex v is included in the slicing criterion, we only follow edges $e = (v', v)$ from the set : $E_{\mathrm{f}} \cup E_{\mathrm{call}} \cup E_{\mathrm{ct}} \cup E_{\mathrm{t}} \cup E_{\mathrm{pi}}$. As we do not follow edges from E_{po}, it will not explore the procedures called by any vertex in the slicing criterion. Indeed, edges in E_{po} are the only edges that allow us to explore a called procedure if we use backward reachability. The called entries will be included in the second phase.

The second phase continues from the set computed in the first phase. But, here, it will only "go down" in the called procedure. In practice, if we have a vertex v in the slice, the second phase will only explore all edges (v', v) from the set $E_{\mathrm{f}} \cup E_{\mathrm{ct}} \cup E_{\mathrm{po}}$. As we don't "go up", we avoid the call context problem.

In [HRB90], Horwitz *et al.* describe one problem that appears when using this technique. Indeed, consider the program of figure 4.5; its dependence graph is shown in figure 4.6.

```
method Q()                method P()
  x := 4; //a1               y := x; //a2
  P(); //c                end_method
  print(y); // printy
end_method
```

Figure 4.5: Problem of the two phases algorithm

In this case, to compute the value of `y` at `print` instruction, we need to keep the assignment `a1` in the slice. But, if we compute the slicing using the two phases technique presented above, `a1` will not be included. Because, procedure P will only be included in the second phase. But, in that phase, we are not following edges in E_{pi} anymore. Thus, instruction `a1` will not be included in the slice. To take `a1` into account, we need to compute the *effect* of calling `P` on the flow dependences in `Q`, i.e. considering the call to `P` as a pseudo instruction. To cope with these problems, the notion of summary edges was introduced in [HRB90]. These

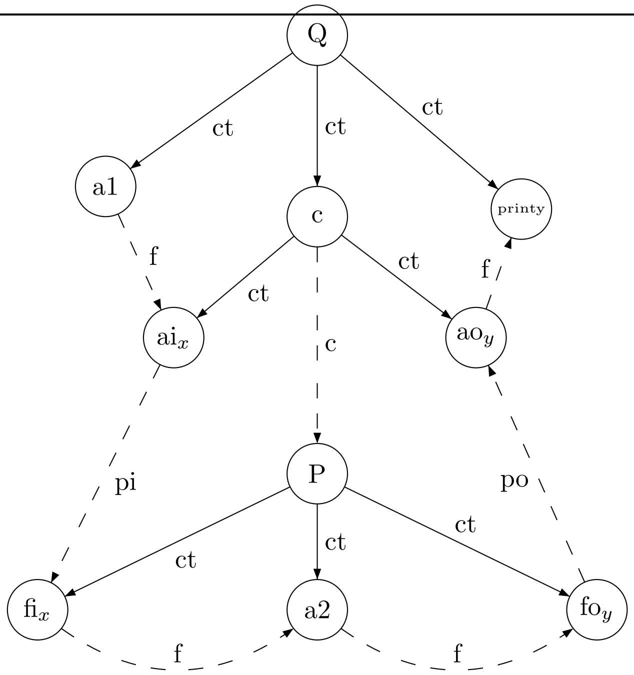

Figure 4.6: Simplified version of the dependence graph of figure 4.5

are a kind of edges that are inserted from vertices in V_{ai} to vertices in V_{ao} . They give a *summary* of the dependences introduced by the corresponding called procedure. The next section tells how these edges are computed

4.3.3 Summary Edges

To have a correct slice when using the 2 phases algorithm described in section 4.3.2, we need to compute summary edges, giving the dependences introduced by calling a given procedure. We now give a quick explanation on a procedure presented in [RHSR94] to compute these edges. This procedure will, for each call, and for each variable, analyse what may be needed by the called procedure to compute the value of that variable. For this, it starts by exploring all intra procedural dependences. This exploration leads to the creation of *path edges*, these edges *sum up* intra procedural dependences. Then, everytime we have found a new (transitive) dependence (i.e. path edge) from a `formal in` vertex v to a `formal out` vertex v' (for the variable analysed) in the considered entry, we propagate this dependence to all potential callers/triggerers (more precisely, to the corresponding `actual in` and `actual out` vertices). Indeed, this kind of dependence tells that the value of the variable defined by v (i.e. before the execution of the entry) may be needed to determine the value of the variable used at v' (i.e. after the execution of the entry). Actually, this procedure performs a kind of intra procedural slicing to discover these dependences. Then, it propagates these discovered dependences to all callers/triggerers. As this propagation may introduce new dependences on the calling side, we need to reconsider all possible dependences in procedures containing the call. This procedure iterates until no new dependence is found. The algorithm can be found in [RHSR94]. Figure 4.7 show the dependence graph with summary edges computed. Unfortunately, this technique of summary edges may still introduce over-conservative dependences. Next section presents a problem that we discovered during the implementation of the algorithm of [RHSR94].

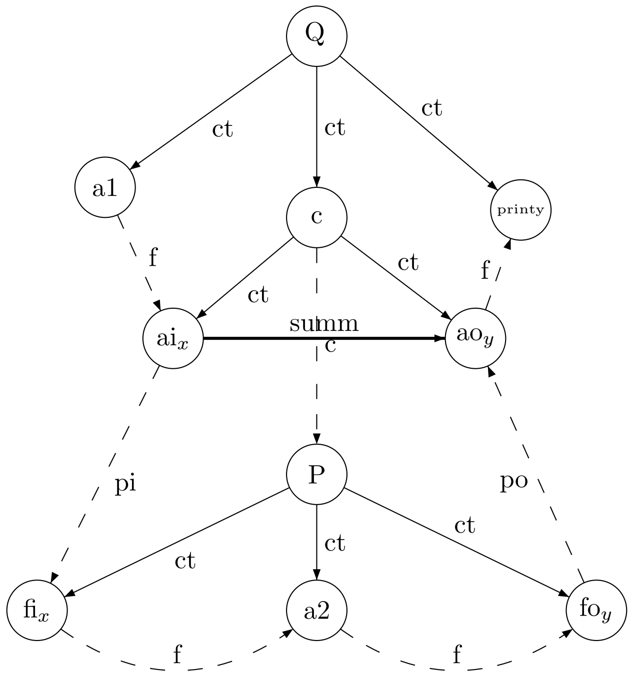

Figure 4.7: Simplified dependence graph for figure 4.5 - bold edges represent summary edges

4.3.4 Problem with summary edges

Let's consider the $_{\mathrm{d}}$SL program of figure 4.8 with instruction `print(y)` as slicing criterion, figure 4.9 gives a simplified version of its SDG. If we compute the summary edges on this program, we will discover, for `Q` that there is a *path edge* from the `formal in` vertex for variable z to the `formal out` vertex for any other variable (e.g. y). Because the call `cp` is control dependent on the `if` that uses variable z and fake vertices and the corresponding flow dependences for call `cp` are introduced in the SDG. As we can see on figure 4.9, `a` will be included in the slice. The problem here is that, in [RHSR94], they are considering path edges, even if they do not contain any *real* definition of the variable analysed (i.e. the variable of the `formal out` vertex). Indeed, we should only consider insertion of summary edges only if, when executing the called/triggered entry, we may meet a *real* definition of the variable analysed. If no *real* definition could be found, then we should "skip" this call.

We now formalise this condition. First, definitions 4.15 and 4.16 define what is a *real* `actual out` vertex. Then, with definitions 4.17 and 4.18, we explain how to compute the set of real `actual out` vertices. Lemma 4.2 proves that these definitions are equivalent.

Once the notion of real `actual out` vertex is defined, we introduce, in definition 4.21, the notion of synchronous correct path. Then, definition 4.22 and 4.23 present a method to compute the set of summary edges respecting this notion of synchronous correct path. Proposition 4.1 proves that these definitions are equivalent.

```
when x == 3 then //w1        method Q()                method P()
  z := 9;//a                   if(z==3) then //if        (*does not modify global vars*)
  Q();  //cq                       P(); //cp          end_method
  print(y); //printy             end_if
end_when                     end_method
```

Figure 4.8: Problem with summary edges

Definition 4.15 - Useful Call
Let $G = (V, E)$ be a dependence graph of a CFG C of a $_{\mathsf{d}}$SL program P, $c \in V_n$ be a call/trigger of P, and let $x \in Var_A(P)$. We say that c is a *useful call* for x iff $\exists e \in V_{ep}$ and the following holds :

1. $(c, e) \in E_{\mathsf{scall}} \cup E_{\mathsf{ts}}$
2. $\exists fo \in V_{\mathsf{fo}}\ :\ (e, fo) \in E_{\mathsf{ct}} \wedge \text{USE}(fo) = \{x\}$. fo is the `formal out` vertex for x in e.
3. $\exists \rho \in \rho(G)\ :\ \rho \equiv v_0 \to \ldots \to v_n = fo$ such that :
   - $v_0 \in V_{\mathsf{n}} \wedge \text{DEF}(v_0) \ni x$
   - $\forall i \in [1, n]\ :\ (v_{i-1}, v_i) \in \big(E_{\mathsf{ct}} \cup E_{\mathsf{fs}} \cup E_{\mathsf{par}}\big)$. This is a synchronous inter procedural path. I.e. we do not follow asynchronous edges.

Note that only synchronous inter procedural paths are considered in point 3. Indeed, c is a useful call for x if and only if, during the synchronous execution of c, we may encounter an assignment of x. For the asynchronous dependences to be taken into account, we need to go through a message treatment phase; remark that no such phase is performed during the treatment of a `WHEN`.

Definition 4.16 - Real `actual out` vertex
Let $G = (V, E)$ be a dependence graph of a CFG C of a $_{\mathsf{d}}$SL program P, $c \in V_n$ be a call/trigger of P, and let $ao \in V_{\mathsf{ao}}$, we have that ao is a *real* `actual out` vertex iff $\text{NORM}(ao)$ is a useful call for $\text{DEF}(ao)$.

ao is a *real* `actual out` vertex if variable $\text{DEF}(ao)$ may be defined during an execution of the corresponding call (i.e. $\text{NORM}(ao)$).

These *real* `actual out` vertices will be useful in the future, and we now present a function computing this set of vertices.

Definition 4.17 - REAL function
Let $G = (V, E)$ be a dependence graph of a CFG C of a $_{\mathsf{d}}$SL program P, $R \subseteq V_{\mathsf{ao}}$ be a set of `actual out vertices`. For G, we define the function $\text{REAL} :\ 2^{V_{\mathsf{ao}}} \to 2^{V_{\mathsf{ao}}}$ such that $ao \in \text{REAL}(R)$ iff $ao \in R \vee \exists \rho \equiv v_0 \to \ldots \to v_n = fo$ and

- $(fo, ao) \in E_{\mathsf{po}}$.
- $\forall i \in [1, n]\ :\ (v_{i-1}, v_i) \in E_{\mathsf{fs}} \cup E_{\mathsf{ct}}$. This is an intra procedural path.

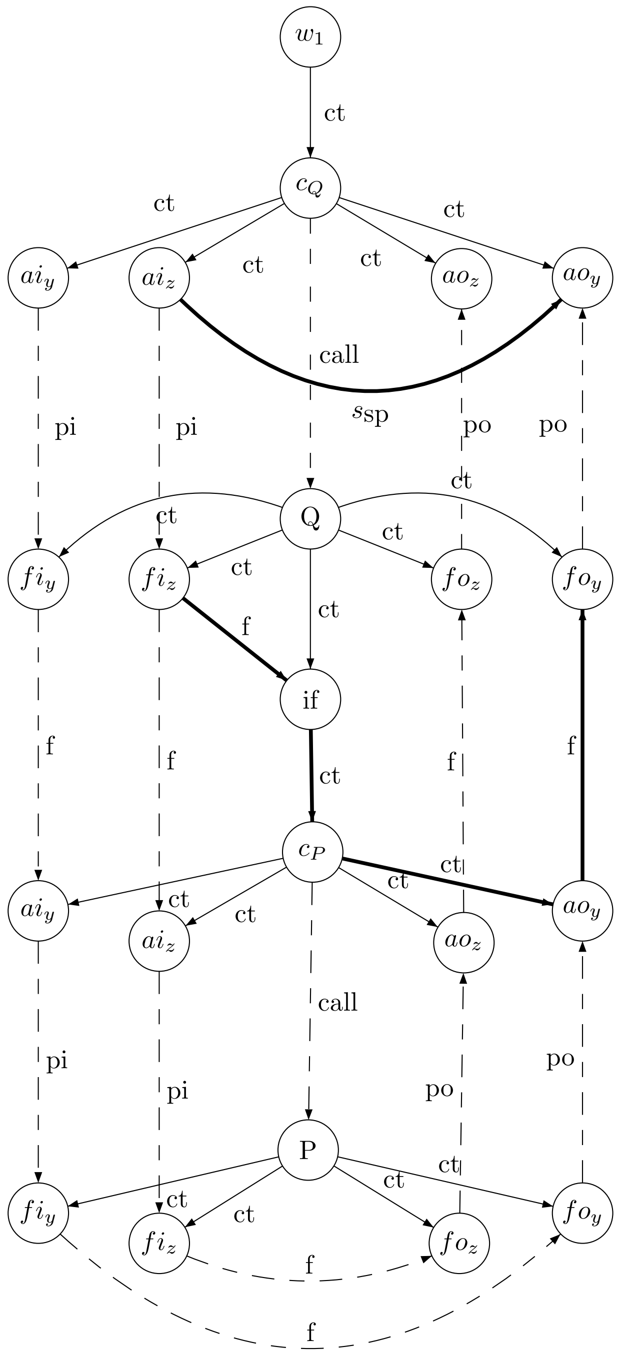

Figure 4.9: Simplified version of the SDG of the $_{\text{d}}$SL program of figure 4.8. Here, bold edges represent a path edge that will lead to the insertion of s_{sp} as summary edges.

- $\text{DEF}(v_0) = \text{USE}(fo) \wedge v_0 \in (V_n \cup R)$. We can meet a *real* definition of $\text{USE}\ (fo)$.

  Note that ρ can be computed using classical algorithms on graph.

Using this function, we may define the set R.

Definition 4.18 - R set
Let $G = (V, E)$ be a dependence graph of a CFG C of a $_{\mathsf{d}}$SL program P. We define the set $R \subseteq V_{\mathsf{ao}}$ as the least fixpoint of function REAL :

$$R = \mu X \cdot \text{REAL}(X)$$

As the size of V_{ao} is finite, we have that R is of finite size. We will prove later that this set R corresponds to the set of real `actual out` vertices.
Before detailing the paths that should be considered when computing summary edges, we first introduce two concepts regarding calls included in this path.

Definition 4.19 - Call level function - cl
Let $G = (V, E)$ be a dependence graph of a CFG C of a $_{\mathsf{d}}$SL program P, and $\rho \equiv v_0 \to \ldots \to v_n \in \rho(G)$ be a path in G. We define the partial function call level, noted $cl_\rho\ :\ \mathbb{N} \to \mathbb{N}$, as follows : $cl_\rho(0) = 0$ and

$$cl_\rho(i) = \begin{cases} cl_\rho(i-1)+1 & \text{if } v_i \in V_{\mathsf{ep}} \cup V_{\mathsf{fi}} \text{ We are exploring a call/trigger} \\ cl_\rho(i-1)-1 & \text{if } v_i \in V_{\mathsf{fo}} \text{ We are leaving a call/trigger} \\ cl_\rho(i-1) & \text{otherwise} \end{cases}$$

Definition 4.20 - Corresponding entry - ce, corresponding exit - cx
Let $G = (V, E)$ be a dependence graph of a CFG C of a $_{\mathsf{d}}$SL program P, and $\rho \equiv v_0 \to \ldots \to v_n \in \rho(G)$ be a path in G. We define the partial functions $ce_\rho\ :\ \mathbb{N} \to \mathbb{N}$ and $cx_\rho\ :\ \mathbb{N} \to \mathbb{N}$ as follows : $ce_\rho(i) = j$ iff :

$$j \leq i \wedge\ cl_\rho(j) = cl_\rho(i) \wedge v_j \in V_{\mathsf{fi}} \cup V_{\mathsf{ep}} \wedge \forall k \in [j, i]\ :\ cl_\rho(k) \geq cl_\rho(i)$$

Moreover, $cx_\rho(i) = j$ iff :

$$j \geq i \wedge\ cl_\rho(j) = cl_\rho(i) \wedge v_j \in V_{\mathsf{fo}} \wedge \forall k \in [i, j]\ :\ cl_\rho(k) \geq cl_\rho(i)$$

The two previous definitions are used to work on the "call stack" of a path. By using the function cl, ce, cx, we can isolate the entry point, the exit point, and the call level of a given vertex in the path.

Example 4.1
Figure 4.10 gives an example for functions of the two previous definitions. Let $\rho \equiv v_0 \to v_1 \to \ldots \to v_{16}$, we have the following :

- $v_0, v_7, v_6 \in V_{\mathsf{fi}}$, $v_{16}, v_{12}, v_9 \in V_{\mathsf{fo}}$, $v_3, v_6 \in V_{\mathsf{ai}}$, $v_{10}, v_{13} \in V_{\mathsf{ai}}$
- $ce_\rho(7) = ce_\rho(8) = ce_\rho(9) = 7$,
  $cx_\rho(7) = cx_\rho(8) = cx_\rho(9) = 9$
- $ce_\rho(4) = ce_\rho(5) = ce_\rho(6) = ce_\rho(10) = ce_\rho(11) = ce_\rho(12) = 4$,
  $cx_\rho(4) = cx_\rho(5) = cx_\rho(6) = cx_\rho(10) = cx_\rho(11) = cx_\rho(12) = 12$
- $ce_\rho(0) = ce_\rho(1) = ce_\rho(2) = ce_\rho(3) = ce_\rho(13) = ce_\rho(14) = ce_\rho(15) = ce_\rho(16) = 16$,
  $cx_\rho(0) = cx_\rho(1) = cx_\rho(2) = cx_\rho(3) = cx_\rho(13) = cx_\rho(14) = cx_\rho(15) = cx_\rho(16) = 0$

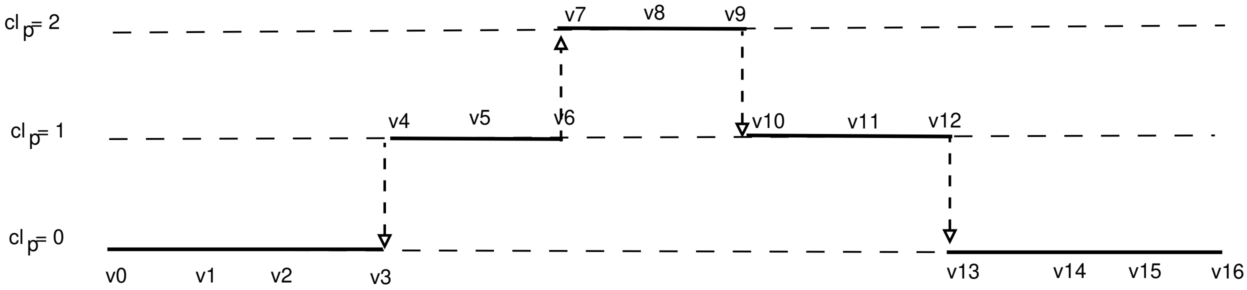

Figure 4.10: Example for functions cl_ρ

Definition 4.21 - Synchronous Correct Path

Let $G = (V, E)$ be a dependence graph of a CFG C of a dSL program P, and $\rho \equiv ai = v_0 \rightarrow \ldots \rightarrow v_n = ao \in \rho(G)$ be a path in G, such that $ai \in V_{\mathsf{ai}}$, $ao \in V_{\mathsf{ao}}$ and $\text{NORM}(ai) = \text{NORM}(ao)$. We say that ρ is a synchronous correct path iff

1. $v_{n-1} \in V_{\mathsf{fo}}$
2. $\forall i \in [1, n] \ : \ (v_{i-1}, v_i) \in \left(E_{\mathsf{ct}} \cup E_{\mathsf{fs}} \cup E_{\mathsf{par}} \cup E_{\mathsf{scall}}\right)$. This path follows only synchronous edges (control, synchronous flow, and parameter edges corresponding to synchronous calls).
3. $\forall i \in [1, n] \ : \ v_i \in V_{\mathsf{ao}} \wedge (v_{i-1}, v_i) \in E_{\mathsf{po}} \Rightarrow$
   $\exists j \in [ce_\rho(i-1), i-1] \ : \ v_j \in V_{\mathsf{n}} \wedge \text{DEF}(v_j) = \text{DEF}(v_i) \ \wedge \exists fo_1, \ldots fo_m \in [j, i] \ :$
   - $fo_1 = cx_\rho(j)$
   - $fo_m = i - 1$ (i.e. $(v_{fo_m}, v_i) \in E_{\mathsf{po}}$) $\wedge$ $v_{fo_m} \in V_{\mathsf{fo}}$
   - $\forall k \in [1, m-1] \ : \ (fo_{k+1} = cx_\rho(fo_k + 1) \wedge \text{USE}(v_{fo_k}) = \text{USE}(v_{fo_{k+1}}) \wedge v_{fo_k} \in V_{\mathsf{fo}})$

ρ is a synchronous correct path for $\text{DEF}(ao)$ iff, for calls/triggers encountered in ρ, either we "skip" them (i.e. $(v_{i-1}, v_i) \notin E_{\mathsf{po}}$ in condition 3), or we can find the corresponding assignment in ρ. Condition 3 also ensures that this assignement should be taken into account at the v_i vertex. For this, every `formal out` vertex that appears in the path from this assignment to the v_i vertex should be related to this variable. In point 3 of definition, we consider the series of fo_k in ρ, such that these fo_k are the `formal out` vertices in the "call path" from v_j to v_i. Each of these `formal out` vertices should use the variable defined by v_j. We need this condition to be sure that we do not explore spurious paths.

We now can identify where summary edges should be inserted. Indeed, we should insert summary edges ending at an `actual out` vertex only if we are considering synchronous correct paths for the corresponding variable.

To give a clear definition of summary edges, we first introduce a fonction, called SUM defined as follows :

Definition 4.22 - SUM function

Let $G = (V, E)$ be a dependence graph of a CFG C of a dSL program P, $S \subseteq 2^{V \times V}$ be a set of edges and $R = \mu X \cdot \text{REAL}(X)$ be the least fixpoint of the function REAL. We define the function $\text{SUM} : \ 2^{V \times V} \rightarrow 2^{V \times V}$ such that $(ai, ao) \in \text{SUM}(S)$ iff $(ai, ao) \in S \vee \exists \rho \equiv fi = v_0 \rightarrow \ldots v_n = fo \in \rho(G)$ and

- $\text{NORM}(ai) = \text{NORM}(ao)$.
- $(ai, fi) \in E_{\mathsf{pi}} \wedge (fo, ao) \in E_{\mathsf{po}}$.
- $\forall i \in [1, n] \ : \ (v_{i-1}, v_i) \in E_{\mathsf{fs}} \cup E_{\mathsf{ct}} \cup S$. This is an intra procedural path.
- $\exists i \in [1, n] \ : \ \text{DEF}(v_i) = \text{USE}(fo) \wedge (v_i \in V_{\mathsf{n}} \vee (v_i \in R \wedge (v_{i-1}, v_i) \in S \cup E_{\mathsf{ct}}))$. We can meet a definition of USE (fo), and if it is in a call, we must include it. Indeed, if $(v_{i-1}, v_i) \in E_{\mathsf{fs}}$, then we "skip" this call (cfr. definition 4.6).
- $\forall i \in [1, n] \ : \ v_i \in V_{\mathsf{ao}} \wedge (v_{i-1}, v_i) \in E_{\mathsf{ct}} \cup S \Rightarrow v_i \in R$. We always have useful calls in the path, or we "skip" this call.

Note that ρ can be computed using classical algorithms on graphs.

Definition 4.23 - summary edges (revisited)
We define the set S of summary edges as the least fixpoint of function SUM :

$$S = \mu X.\text{SUM}(X)$$

As the number of `actual in/out` vertices is finite, we have that S is of finite size.

We now prove that this definition of summary edges is equivalent to the definition of synchronous correct path. Thus a procedure computing summary edges following previous definition would respect the condition on synchronous correct paths. For this, we need two new lemmas.

Lemma 4.1
Let $G = (V, E)$ be a dependence graph of a CFG C of a $_{\mathsf{d}}$SL program P, and let $v \in V_{\mathsf{n}} \setminus V_{\mathsf{ep}}$ be a vertex of G. We have that $\exists \rho \equiv \text{ENTRYOF}(v) = v_0 \to \ldots \to v_n = v \in \rho(G)$ with $\forall i \in [1, n] \ : \ (v_{i-1}, v_i) \in E_{\mathsf{ct}}$. ■

This lemma simply tells that, for any given normal vertex, we have a path starting from its entry point to this vertex following only control edges. Indeed, during the construction of the set E_{ct}, control edges were added to keep the structure of the original $_{\mathsf{d}}$SL program. By following these edges, we can reach any instruction of a given entry point.

The next lemma states the equivalence between the R set and the real `actual out` vertices.

Lemma 4.2
Let $G = (V, E)$ be a dependence graph of a CFG C of a $_{\mathsf{d}}$SL program P, let $R = \mu X.\text{REAL}(X)$. We have that $\forall ao \in V_{\mathsf{ao}} \ : \ ao \in R \Leftrightarrow ao$ is a real `actual out` vertex.

Proof
To prove this result, we define $R^i = \text{REAL}^i(\emptyset)$ and let m be the least natural such that $R^m = R^{m+1}$. Hence, we have that $\mu X.\text{REAL}(X)$ computes R^m.

$\boxed{\Rightarrow}$

We have a vertex $ao \in R$, we must prove that this vertex is a real `actual out` vertex. For this, we will use definition 4.17 of function REAL to construct a path ending at v. Then, we prove that this path respects the conditions of definition 4.15 for useful calls.

We first introduce the procedure `ConstructUsefulCall`. This procedure takes as parameters the natural m defined above and a path starting from a vertex in R. Then, it expands this path to have a path respecting the conditions of definition 4.15 for useful calls.

Procedure `ConstructUsefulCall`(ρ, i) - Expanding procedure

Data : $i \in \mathbb{N},\ \rho \in \rho(G)$ such that $\rho \equiv ao \cdot \rho'$ and $ao \in R^i$
Result: $\rho_u \in \rho(G)$
begin
We have $\rho \equiv ao \cdot \rho'$;
We have $\rho'' = v_0 \to \ldots \to v_n = fo \wedge v_0 \in (V_{\mathsf{n}} \cup R^{i-1}) \wedge \mathrm{DEF}(v_0) = \mathrm{USE}(fo) = \mathrm{DEF}(ao)$ (as $ao \in R^i$, by def 4.22);
if $v_0 \in V_n$ **then**
$\rho_u \leftarrow \rho'' \cdot \rho$;
else
$\rho_u \leftarrow$ `ConstructUsefulCall`$(\rho'' \cdot \rho, i-1)$;
end
return ρ_u;
end

We have that $\forall \rho \equiv v \cdot \rho'$ with $v \in R^m$, `ConstructUsefulCall`(ρ, m) computes a path ρ_u in a finite number of steps. Indeed, in the worst case, we can have a call `ConstructUsefulCall`$(\rho'', 0)$. But, as m is a natural, we would reach this call in a finite number of steps.

Let's compute $\rho_u \equiv$ `ConstructUsefulCall`(ao, m). In order to prove that ao is a real `actual out` vertex, we must prove that ρ_u respects conditions of definition 4.15 of useful calls.

Let $\rho_u \equiv v_0 \to \ldots \to v_n$, ρ_u is such that $v_n = ao$ and $\forall i \in [1, n](v_{i-1}, v_i) \in (E_{\mathsf{ct}} \cup E_{\mathsf{fs}} \cup E_{\mathsf{par}})$. Moreover, $\mathrm{DEF}(v_0) = \mathrm{DEF}(ao)$. From procedure `ConstructUsefulCall` presented above, it is easy to see that $v_0 \in V_{\mathsf{n}}$.
In conclusion, we have that ao is a real `actual` out vertex.

$\boxed{\Leftarrow}$
We have that ao is a real `actual out` vertex, and we want to prove that $ao \in R$. If ao is a real `actual out` vertex, then, by definition 4.15 of useful calls, we have a path ρ respecting conditions of definition 4.15. We prove by induction on the *call depth* of ρ (defined below) that ao is included during the computation of R.

If ao is a real `actual out` vertex, we have, by definition 4.15 that $\exists \rho \equiv v_0 \to \ldots \to v_n = fo \to ao$.
We have that $v_0 \in V_{\mathsf{n}}$, $\mathrm{DEF}(v_0) = \mathrm{DEF}(ao)$, and $\forall i \in [1, n]\ :\ (v_{i-1}, v_i) \in (E_{\mathsf{ct}} \cup E_{\mathsf{fs}} \cup E_{\mathsf{par}})$. Let's define the partial function $cd : \rho(G) \to \mathbb{N}$ such that

$$cd(\rho' \equiv v_0 \to \ldots \to v_m) = |\{i | i \in [1, m] \wedge v_i \in V_{\mathsf{ao}} \wedge (v_{i-1}, v_i) \in E_{\mathsf{par}}\}|$$

Informally, this function returns the number of calls that had to be explored to reach v_0. We prove that $ao \in R$ by induction on $cd(\rho)$.

$cd(\rho) = 1$

We have a path $\rho \equiv v_0 \to \ldots \to v_n = fo \to ao$ and $\forall i \in [1, n] \; : \; (v_{i-1}, v_i) \in E_{\mathsf{fs}} \cup E_{\mathsf{ct}}$. By definition of ρ, we have that $v_0 \in V_{\mathsf{n}} \wedge \textsc{def}(v_0) = \textsc{def}(ao)$. In conclusion, this path respects conditions of definition 4.17. Thus, we have that $ao \in R$.

$cd(\rho) = k > 1$

We define $j \in [1, n-1]$ such that $v_j \in V_{\mathsf{ao}} \wedge (v_{j-1}, v_j) \in E_{\mathsf{par}}$ and $\forall i \in [j, n] \; : \; (v_j \notin V_{\mathsf{ao}} \vee (v_{j-1}, v_j) \in E_{\mathsf{fs}} \cup E_{\mathsf{ct}})$. I.e. j is the last `actual out` vertex such that the corresponding call is explored in ρ.
If we consider the path $\rho' \equiv v_0 \to \ldots \to v_j$, we have that $cd(\rho') = k - 1$ and, by the induction hypothesis, we can prove that $v_j \in R$.
Let's now consider the path $\rho'' \equiv v_j \to \ldots \to v_n = fo \to ao$. By definition of j, we have that ρ'' is an intra procedural path. Moreover, as, $v_j \in R$ we have that ρ'' respects the conditions of definition 4.22. Thus, we have that $ao \in R$.

In conclusion, we have that $ao \in R$. ■

We now give the lemma that will be used to design a procedure for computing summary edges.

Proposition 4.1
Let $G = (V, E)$ be a dependence graph of a $\textsc{cfg}$ C of a $_{\mathsf{d}}$SL program P, and let $S = \mu X.\textsc{sum}(X)$. We have that $\forall ai \in V_{\mathsf{ai}}, \; ao \in V_{\mathsf{ao}}$:

$$(ai, ao) \in S \Leftrightarrow \exists \rho \equiv ai \to fi \to \ldots \to fo \to ao \in \rho(G) \; : \; \rho \text{ is a synchronous correct path}$$

Proof
Let $R = \mu X.\textsc{real}(X)$ be the least fixpoint of function $\textsc{real}$. To prove this result, we define $S^i = \textsc{sum}^i(\emptyset)$ and let m be the least natural such that $S^m = S^{m+1}$. Hence, we have that $\mu X.\textsc{sum}(X)$ computes S^m.

$\boxed{\Rightarrow}$
We have an edge $(ai, ao) \in S$, and we must prove that there is a synchronous correct path (definition 4.21) starting from ai and ending at ao. For this, we use definition 4.22 of summary edges to construct this path.

We first introduce the procedure `ConstructCorrectPath` that takes as parameters the natural m defined above and the edge (ai, ao). Using definition 4.22 and definition 4.17, this procedure starts from this edge and build a complete path. Then, we prove that this path is a synchronous correct path.

Procedure `ConstructCorrectPath(`ρ, l`)` - Expanding procedure

Data : $\rho \in \rho(G),\ l \in nat$
Result: $\rho_c \in \rho(G)$
begin
  Let $\rho \equiv v_0 \to \ldots \to v_n$;
  Let $\rho_c \equiv \rho$;
  for *each* $i \in [1, n]\ :\ v_i \in V_{ao}$ **do**
    We have $\rho_c \equiv \rho_{bc} \cdot v_i \cdot \rho_{ac}$;
    if $(v_{i-1}, v_i) \in S^l$ *(i.e.* $v_{i-1} \in V_{ai}$*)* **then**
      Let $\rho' \equiv fi = v_0 \to \ldots \to v_n = fo$ (as $(v_{i-1}, v_i) \in S^l$, by definition 4.22) such that :

      - $(ai, fi) \in E_{\text{pi}} \wedge (fo, ao) \in E_{\text{po}}$. Thus, we have $\text{DEF}(ao) = \text{USE}(fo)$.
      - $\forall i \in [1, n]\ :\ (v_{i-1}, v_i) \in E_{\text{fs}} \cup E_{\text{ct}} \cup S^{l-1}$
      - $\exists i \in [1, n]\ :\ \text{DEF}(v_i) = \text{USE}(fo) \wedge \big(v_i \in V_{\text{n}} \vee (v_i \in R \wedge (v_{i-1}, v_i) \in S^{l-1} \cup E_{\text{ct}})\big)$
      - $\forall i \in [1, n]\ :\ v_i \in V_{\text{ao}} \wedge (v_{i-1}, v_i) \in E_{\text{ct}} \cup S^{l-1} \Rightarrow v_i \in R$

      ;
      Let $\rho'_c \leftarrow$`ConstructCorrectPath(`$\rho', l - 1$`)`;
    end
    if $(v_{i-1}, v_i) \in E_{ct}$ *(i.e.* v_{i-1} *is a call/trigger)* **then**
      By definition 4.22, $v_i \in R$. We can apply procedure `ConstructUsefulCall()` on vi;
      Let $\rho_u \equiv$ `ConstructUsefulCall(`v_i, m`)`, $\rho_u \equiv v'_0 \to \ldots \to v'_m = v_i$;
      We have that $v'_0 \in V_{\text{n}} \wedge \text{DEF}(v'_0) = \text{DEF}(v_i)$;
      Let $\rho_{call} \equiv \text{ENTRYOF}(v'_0) \to \ldots \to v'_0$(by lemma 4.1);
      Let ρ'_c such that $\rho_{call} \equiv \rho'_c \cdot v'_0$;
      for *each* $i \in [1, m]\ :\ v'_i \in V_{ao}$ **do**
        Let $c = \text{NORM}(v'_i)$;
        Let $\rho_{ct} \equiv \text{ENTRYOF}(c) \to \ldots \to c$ (by lemma 4.1);
        $\rho'_c \leftarrow \rho_{ct} \cdot \rho'_c \cdot \rho_u$;
      end
    end
    $\rho_c \leftarrow \rho_{bc} \cdot \rho'_c \cdot v_i \cdot \rho_{ac}$;
  end
  return ρ_c;
end

Let's consider $\rho_c \equiv$`ConstructCorrectPath(`$(ai, ao), m$`)`. ρ_c will be computed in a finite number of steps. Indeed, in the worst case, we have a call to `ConstructCorrectPath(`$\rho, 0$`)`. But, as m is finite, this call will be reached in a finite number of steps. Moreover, the length of each path generated by this procedure is finite. Thus, only a finite number of iterations is performed at each call.

We must prove that ρ_c respects conditions of definition 4.21 of a synchronous correct path.

Let $\rho_c \equiv ai = v_0 \to \ldots \to v_n = ao$. We have that ρ_c is a synchronous correct path for $\text{DEF}\ (ao)$. Indeed :

- $v_{n-1} \in V_{\mathsf{fo}}$
- $\forall i \in [1,n] \; : \; (v_{i-1}, v_i) \in \big(E_{\mathsf{ct}} \cup E_{\mathsf{fs}} \cup E_{\mathsf{par}} \cup E_{\mathsf{scall}}\big)$
- We must prove that $\forall i \in [1,n] \; : \; v_i \in V_{\mathsf{ao}} \wedge (v_{i-1}, v_i) \in E_{\mathsf{po}} \Rightarrow$
  $\exists j \in [ce_\rho(i-1), i-1] \; : \; v_j \in V_{\mathsf{n}} \wedge \mathrm{DEF}(v_j) = \mathrm{DEF}(v_i) \; \wedge \exists fo_1, \ldots fo_m \in [j,i] \; :$
  - $fo_1 = cx_\rho(j)$
  - $fo_m = i-1$ (i.e. $(v_{fo_m}, v_i) \in E_{\mathsf{po}}$)
  - $\forall k \in [1, m-1] \; : \; (fo_{k+1} = cx_\rho(fo_k + 1) \wedge \mathrm{USE}(fo_k) = \mathrm{USE}(fo_{k+1}) \wedge v_{fo_k} \in V_{\mathsf{fo}})$
  - $v_{fo_m} \in V_{\mathsf{fo}}$

  We proceed by contradiction, let's suppose that

$$\exists i \in [1,n] \; : \; v_i \in V_{\mathsf{ao}} \wedge (v_{i-1}, v_i) \notin E_{\mathsf{fs}} \wedge \nexists j \in [ce_\rho(i-1), i-1] \; : \; v_j \in V_{\mathsf{n}} \wedge \mathrm{DEF}(v_j) \neq \mathrm{DEF}(v_i) \quad (4.1)$$

We have that v_{i-1} is a `formal out` vertex linked to v_i. $v_{ce_\rho(i-1)}$ is the corresponding entry vertex and is an entry point or a `formal in` vertex.

If $v_{ce_\rho(i-1)} \in V_{\mathsf{ep}}$, then `ConstructCorrectPath` called the procedure `ConstructUsefulCall`. By calling this procedure, we have inserted in ρ_c a vertex $v \in V_{\mathsf{n}}$ such that $\mathrm{DEF}(v) = \mathrm{DEF}(v_i)$ (see proof of lemma 4.2). Moreover, v was inserted in $[ce_\rho(i-1), i-1]$ Thus, we have a contradiction.

If $v_{ce_\rho(i-1)} \in V_{\mathsf{fi}}$, then we have $(v_{ce_\rho(i-1)-1}, v_i) \in S$. Using the hypothesis and definition 4.22, we can deduce that

$$\exists j \in [ce_\rho(i-1), i] \; : \; \mathrm{DEF}(v_j) = \mathrm{DEF}(v_i) \wedge (v_j \in V_{\mathsf{n}} \vee (v_j \in R \subseteq V_{\mathsf{ao}} \wedge (v_{j-1}, v_j) \in S \cup E_{\mathsf{ct}}))$$

By equation 4.1, we know that $v_j \notin V_{\mathsf{n}}$. Thus, we have that $v_j \in V_{\mathsf{ao}} \wedge (v_{j-1}, v_j) \in S$. But, in this case, procedure `ConstructCorrectPath` would have continued to expand this edge. As we know that `ConstructCorrectPath`$((ai, ao), m)$ has been computed in a finite number of steps, we have a contradiction.

We know that

$$\forall i \in [1,n] \; : \; v_i \in V_{\mathsf{ao}} \wedge (v_{i-1}, v_i) \notin E_{\mathsf{fs}} \wedge \exists j \in [ce_\rho(i-1), i] \; : \; v_j \in V_{\mathsf{n}} \wedge \mathrm{DEF}(v_j) = \mathrm{DEF}(v_i)$$

Let's look at an `actual out` vertex $v_i \in V_{\mathsf{ao}}$ in ρ_c such that $(v_{i-1}, v_i) \notin E_{\mathsf{fs}}$. In `ConstructCorrectPath`, before expansion, we had the edge $(v_{ce_\rho(i-1)-1, v_i})$ in ρ_c. If we examine procedure `ConstructCorrectPath`, in both cases, we insert before v_i a path ending by a `formal out` vertex fo such that $\mathrm{USE}(fo) = \mathrm{DEF}(v_i)$. Furthermore, we have a vertex v in this path such that $\mathrm{DEF}(v) = \mathrm{USE}(fo) = \mathrm{DEF}(v_i)$ and $(v \in V_{\mathsf{n}} \vee (v \in R \wedge \exists (v', v) \in S \cup E_{\mathsf{ct}}))$. After having performed all expansions, we have a vertex $v_j \in V_{\mathsf{n}}$, $\mathrm{DEF}(v_j) = \mathrm{DEF}(v_i)$. And all fo vertices inserted during this expansion were such that $\mathrm{USE}(fo) = \mathrm{DEF}(v_i)$. Thus, we have that $\exists fo_1, \ldots fo_m \in [j,i] \; :$

- $fo_1 = cx_\rho(j)$
- $fo_m = i-1$ (i.e. $(v_{fo_m}, v_i) \in E_{\mathsf{po}}$)

- $\forall k \in [1, m-1]\ :\ (fo_{k+1} = cx_\rho(fo_k + 1) \wedge \text{USE}(fo_k) = \text{USE}(fo_{k+1}) \wedge v_{fo_k} \in V_{\mathsf{fo}})$
- $v_{fo_m} \in V_{\mathsf{fo}}$

$\boxed{\Leftarrow}$

We have a path $\rho \equiv ai \to fi = v_0 \to \ldots \to v_n = fo \to ao$. Using ρ, we prove that $(ai, ao) \in R$.
Let $M = max_{i\in[0,n]}\ cl_\rho(i)$. We prove that $(ai, ao) \in S$ by induction on M.

$M = 1$

In this case, we have an intra procedural path from fi to fo and, by definition of ρ, we have that $\exists j \in [0, n]\ :\ v_j \in V_{\mathsf{n}} \wedge \text{DEF}(v_j) = \text{DEF}(ao)$. Moreover, we "skip" each call of the path. Thus, this path respects the conditions of definition 4.22. In conclusion, we have that $(ai, ao) \in S$.

$M > 1$

Here, we have an inter procedural path. We first need to build an intra procedural path, then we prove that this path respects conditions of definition 4.22 of summary edges. We first prove that we can build such a path. We define the set C as the set of naturals c such that $v_c \in V_{\mathsf{ao}} \wedge cl_\rho(v_c) = 1 \wedge v_{c-1} \in V_{\mathsf{fo}}$. For each natural $c \in C$, let's define the couple (j, j') of naturals such that $j' = c - 1 \wedge j = ce_\rho(j')$. Informally, C gives the set of "root calls" in ρ. We have the following :

1. We have that $v_j \in V_{\mathsf{fi}} \wedge v_{j'} \in V_{\mathsf{fo}}$.
2. By definition of ρ, $\exists i \in]j, j'[\ :\ v_i \in V_{\mathsf{n}} \wedge \text{DEF}(v_i) = \text{USE}(v_c)$. We have an assignment of the corresponding variable.
3. Thus, v_c is a real `actual out` vertex, and by lemma 4.2, $v_{j'+1} \in R$.
4. If $v_{j-1} \in V_{\mathsf{scall}}$, we have that $(v_{j-1}, v_c) \in E_{\mathsf{ct}}$
5. If $v_{j-1} \in V_{\mathsf{ai}}$, we have that the subpath ρ' of ρ from v_{j-1} to v_c is a synchronous correct path. Moreover, $max_i\ cl_{\rho'}(i) = n - 1$. Thus, by the induction hypothesis, we have that $(v_{j-1}, v_c) \in S$.

We now build an intra procedural path from ρ using fact stated above.

Let $\rho_s \equiv ai \to fi = v_{l_0} \to \ldots \to v_{l_i} \to \ldots \to v_{l_m} = fo \to ao$ be a projection of ρ where only vertices of the root call are considered. More formally, $\forall k \in [0, m]$:

- $cl_\rho(k) \leq 1$.
- $\forall i \in [0, k[\ :\ l_i < l_k$.

Note that $l_0 = 0$ and $l_m = n$. We use ρ_s to prove that $(ai, ao) \in S$. Indeed, we have the following points :

- $(ai, fi) \in E_{\mathsf{pi}} \wedge (fo, ao) \in E_{\mathsf{po}}$.
- $\forall i \in [1, m]\ :\ (v_{l_{i-1}}, v_{l_i}) \in E_{\mathsf{fs}} \cup E_{\mathsf{ct}} \cup S$.
- $\forall i \in [1, m]\ :\ v_{l_i} \in V_{\mathsf{ao}} \wedge (v_{l_{i-1}}, v_{l_i}) \in E_{\mathsf{ct}} \cup S \Rightarrow v_{l_i} \in R$. Indeed, if $v_{l_i} \in V_{\mathsf{ao}} \wedge (v_{l_{i-1}}, v_{l_i}) \in E_{\mathsf{ct}} \cup S$, then $l_i \in C$, and, as stated above, $v_{l_i} \in R$.

It now remains to prove that

$$\exists i \in [1,m] \ : \ \text{DEF}(v_{l_i}) = \text{USE}(fo) = \text{DEF}(ao) \wedge \big(v_{l_i} \in V_{\mathsf{n}} \vee (v_{l_i} \in R \wedge (v_{l_{i-1}}, v_{l_i}) \in S \cup E_{\mathsf{ct}})\big) \tag{4.2}$$

From definition of ρ, we can deduce that :
$\exists j \in [1,n] \ : \ v_j \in V_{\mathsf{n}} \wedge \text{DEF}(v_j) = \text{DEF}(ao) \wedge \exists v_{fo_1}, \ldots v_{fo_o} = fo \ : \ \forall k \in [1,o] \ : \ fo_k \in [j,n] \wedge$

- $fo_1 = cx_\rho(j)$
- $\wedge \ \forall k \in [1, o-1] \ : \ (fo_{k+1} = cx_\rho(fo_k + 1) \wedge \text{USE}(v_{fo_k}) = \text{USE}(v_{fo_{k+1}}) \wedge v_{fo_k} \in V_{\mathsf{fo}})$

If $cl_\rho(j) = 1$, then we have that $\exists j' \in [1,m] \ : \ l_{j'} = j$ and condition 4.2 is respected.

If $cl_\rho(j) > 1$, let's look at $v_{fo_{o-1}}$. We have that $\text{USE}(v_{fo_{o-1}}) = \text{DEF}(v_j) = \text{DEF}(ao)$. Moreover, $fo_o = cx_\rho(fo_{o-1} + 1)$, we know that $v_{fo_{o-1}+1}$ is an `actual out` vertex and, as $v_{fo_o} = fo$, $cl_\rho(fo_{o-1} + 1) = 1 \wedge \text{DEF}(v_{fo_{o-1}+1}) = \text{DEF}(ao)$. Thus, in $\rho_s, \exists ao' \in [1,m] \ : \ v_{l_{ao'}} = v_{fo_{o-1}+1}$. As $(v_{l_{ao'-1}}, v_{l_{ao'}}) \in E_{\mathsf{ct}} \cup S$, we have, that $v_{l_{ao'}} \in R$. Thus, condition 4.2 is respected and $(ai, ao) \in S$.

In conclusion, we have that $(ai, ao) \in S$. ■

Procedure `ComputeSumEdges` in appendix D computes both sets S and R following definitions 4.22 and 4.17. This algorithm is called before performing the reachability in itself. Note that it does not follow edges from $E_{\text{fa}}, E_{\text{call}}, E_{\text{ta}}$ and E_{par}, as these edges correspond to inter procedural dependences. This algorithm is inspired from [RHSR94].

Now, having a SDG G and the set of summary edges S and the real `actual out` vertices computed using algorithm , we may apply the two phase slicing as described above. In next section, we present another problem discovered during the implementation of a $_{\text{d}}$SL slicer.

4.4 Implementing accurate two phases slicing.

In the previous section, we presented a technique to compute a precise slice of a $_{\text{d}}$SL program. The dependences due to global variables were considered as parameter passing. And the parameter passing was simulated by fake assignment "inserted" in the control flow graph. To avoid call context problem, we needed to perform a two phases backward exploration, and for this two phases exploration to work, we needed to compute summary edges.

During our implementation of a $_{\text{d}}$SL slicer, we encountered a problem with the original two phases slicing technique of [HRB90]. Indeed, we had over-conservative slices due to the fake vertices included to treat calls and triggers. This section details this problem and a possible solution using formalisms of the previous section.

4.4.1 Example program

Let's consider the $_{\text{d}}$SL program of figure 4.11, figure 4.12 gives a simplified version of its SDG. In this program, `WHEN w1` may trigger `WHEN w2` which does not modify any variable. Thus, this `WHEN w2` and assignment `a` should not be included in the slice for the instruction `print(y)`. The problem comes from the global variables. Remember that `WHEN` triggering is treated as an usual procedure call, and that global variables

```
when x == 3 then //w1      when z==4 then //w2
  z := 4; //a                 (* does not modify x or y. *)
  print(y); //printy       end_when
end_when
```

Figure 4.11: $_{\rm d}$SL program having an over-conservative slice.

are passed as in/out parameters. To manage dependences due to these in/out parameters, `actual in/out` vertices are inserted, modeling the parameter passing. If we apply definitions presented in section 4.2.3, we see that a flow dependence appears between ao_y and `print(y)`. Thus, `WHEN w2` and assignment `a` will also be included in the slice. This problem will arise for any entry called/triggered before any vertex included in the first phase of the slicing. The problem comes from the fact that, when fake vertices are included, we also include vertices controlling these fake vertices. But, the call corresponding to this vertex may not lead to a modification of the corresponding variable. Indeed, if the called entry does not (indirectly) modify the corresponding variable, we do not have a real flow dependence and we should "skip" this call without including it.

It is easy to see that this corresponds to definition 4.15. When considering an `actual out` vertex, if it is not an real `actual out` vertex, we should "skip" the corresponding call/trigger. Indeed, if this call/trigger is not a useful call for this variable, there is no possible execution of this call/trigger where this variable would be modified.

Let $R = \mu X \cdot \textsc{real}(X)$ and $S = \mu X \cdot \textsc{sum}(X)$. As proved in lemma 4.2, we have that $ao \in R$ if and only if $\textsc{norm}(ao)$ is a useful call for $\textsc{def}(ao)$. In practice, during the backward reachability, when we meet an `actual out` vertex ao, we follow all incoming edges only if $ao \in R$. Else, we only follow incoming synchronous flow edges (i.e., we "skip" this call). The next section presents a reachability algorithm taking into account this condition.

4.4.2 Algorithms for two-phases slicing.

This section presents the algorithms described above; algorithm 1 performs the two phases slicing in itself. It first calls the procedure to compute summary edges. Then, it calls a reachability procedures two times, this corresponds to the two phases described above. Procedure Reachability performs the reachability. It receives in parameter the SDG, the set of starting vertices, the set of edges to explore (depending on the phase explored), the set of summary edges, and the set of real `actual out` vertices.

Algorithm 1: Algorithm for two phases slicing.

Data : $G = (V, E)$ a dependence graph, $S \subseteq V$, the slicing criterion
Result: slicing $V' \subseteq V$ of G w.r.t. S
begin
  $(S, R) \leftarrow \texttt{ComputeSumEdges}(G)$;
  $P_1 \leftarrow \texttt{Reachability}(V, S, E_{\rm f} \cup E_{\rm call} \cup E_{\rm ct} \cup E_{\rm t} \cup E_{\rm pi} \cup S, S, R)$;
  $P_2 \leftarrow \texttt{Reachability}(V, P_1, E_{\rm f} \cup E_{\rm ct} \cup E_{\rm po} \cup S, S, R)$;
  return P_2;
end

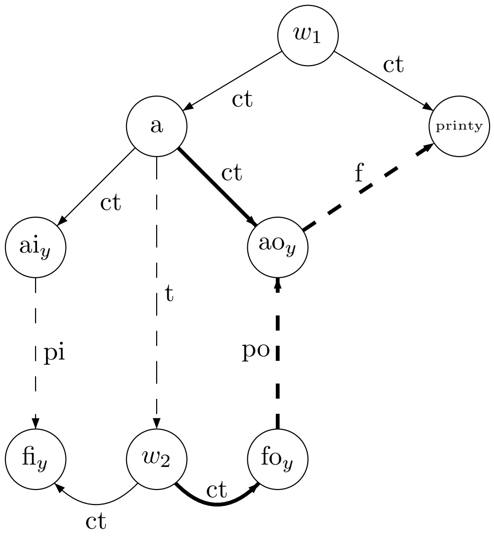

Figure 4.12: Simplified version of dependence graph of figure 4.11. Here, a and w_2 will be included, while not necessary.

Procedure `Reachability(`G, S, E', S, R`)` - reachability function.

Data : $G = (V, E)$ a dependence graph, $S \subseteq V$, set of edges $E' \subseteq E$, S, the set of summary edges, R, the set of real `actual out` vertices

Result: $V' \subseteq V$ - Backward reachability of S in G, following only edges of E'.

begin

worklist $\leftarrow S$;

$V' \leftarrow S$;

mark all $v \in S$ as *explored*;

while *worklist* $\neq \emptyset$ **do**

let $v \in$ *worklist*;

worklist $\leftarrow$ *worklist* $\setminus \{v\}$;

for *each* $e = (v', v) \in E', \neg$ explored(v') **do**

if $v \notin (V_{ao} \setminus R) \ \vee \ e \in E_{fs}$ **then**

mark v' as *explored*;

worklist $\leftarrow$ *worklist* $\cup \{v'\}$;

$V' \leftarrow V' \cup \{v'\}$;

end

return V';

end

4.5 Pratical evaluation of slicing

In this section, we present some tests we performed on the slicer. These tests will be used to evaluate how useful the slicer is during model checking. We first present the different examples and slicing criterion used.

4.5.1 Canal lock controller

This example is an implementation of a canal lock controller. Each lock is 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 of emptied). The different commands of this system (opening/closing a gate, emptying/filling a lock) can be accessed via 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, and (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 any 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 an error.

Figure 4.13 shows a picture of the canal lock controller. Here, we have two slicing criteria. The first one monitors all possible actions that could be performed on one of the two locks (see figure 4.13). Indeed, everytime we performed an action on a gate, or we changed the water level, we inserted the corresponding instruction in the slicing criterion. The second slicing criterion monitors only the two central gates. Indeed, instead of performing a complete model checking, we could only try to check one of the property. In this case, we wanted to check that these gates were never opening at the same time. In table 4.1 "locks sliced" (resp. "locks sliced 2") is the result of the slicing using first (resp. second) slicing criterion. In both cases, the slicing is interesting, as it suppresses, useless code and entry points. Using the second slicing criterion, we have much less `WHEN`'s than in the original code. This proves that, for very specific properties, slicing is useful.

4.5.2 Railway traffic controller

This is a classical example in verification. Two trains share a common critical section and the controller must ensure that there will never be both trains in the critical section at the same time. Figure 4.14 illustrates this problem. We sliced this program with the variable controlling the critical section as slicing criterion. We can see a small gain in the size of the program. The interesting part is the number of sequences that were eliminated. Indeed, during model checking, we won't have to consider all sequences controlling train's motors.

4.5.3 Crane controller

This example is very similar to the railway controller. Two cranes must drop some objects on a common conveyor belt and they must avoid to perform this drop operation at the same time. We have a critical section controlling this conveyor belt The original $_d$SL code is quite heavy. This is due to the fact that it controls all cranes movement (U-turns, dropping). If we slice w.r.t. the critical section, we can suppress all this code and the resulting program is much smaller. We can already see that slicing is useful for $_d$SL, as a lot of operations, useless for the verification of a given property, can be eliminated.

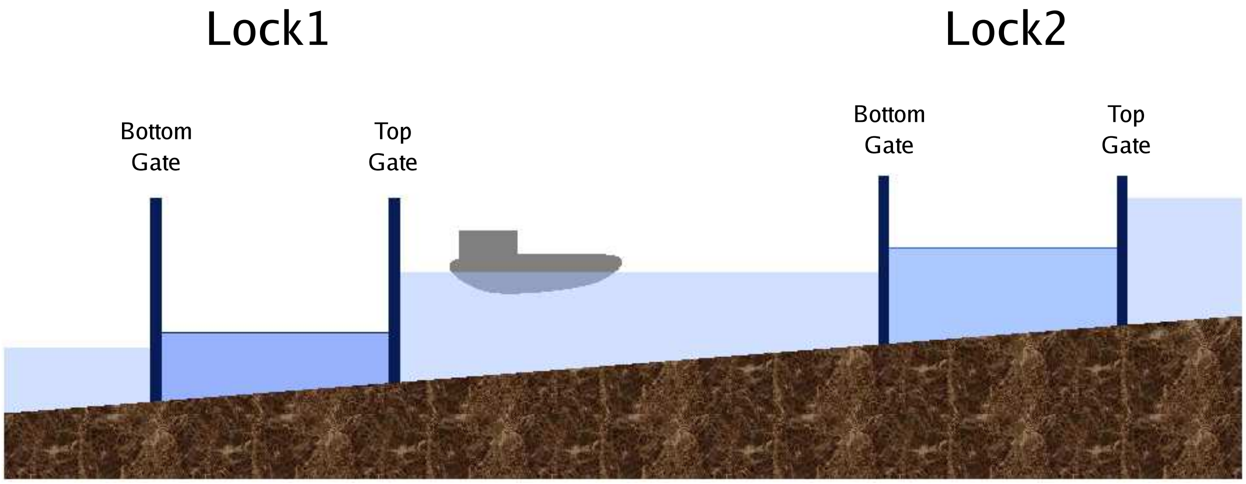

Figure 4.13: Canal lock controller

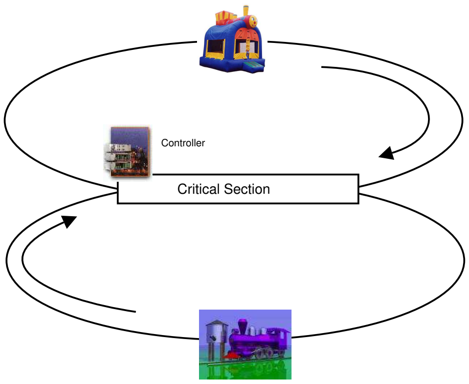

Figure 4.14: Railway traffic controller

4.6 Experimental results

Table 4.1 presents the results of the slicing on the examples presented above. The first column gives the name of the example, and whether it is sliced (and the criterion used). Next column gives the number of line of the program. This line is of the following format : x - y = z, where x is the total size of the program, y the size of the initialisation part and z the result of the substraction. The last three columns give respectively the number of `WHEN`'s, `METHOD`'s and `SEQUENCE`'s of this program.

Example	**lines**	**whens**	**sequences**	**methods**
locks non sliced	270 - 69 = **201**	18	7	0
locks sliced	196 - 69 = **127**	14	5	0
locks sliced 2	162 - 69 = **93**	8	5	0
traffic non sliced	101 - 43 = **58**	6	7	0
traffic sliced	89 - 43 = **46**	6	3	0
Crane non sliced	279 - 45 = **234**	3	17	0
Crane sliced	99 - 45 = **44**	3	5	0

Table 4.1: Tests on slicing.

Chapter 5

Conclusion

In this document, we presented $_{d}$SL, a language to control industrial systems. This language is event-driven and distributed. It is event-driven because some entries may be executed whenever a given event happens. It is distributed because the controller may be composed of different sites. Moreover $_{d}$SL has some mechanisms to allow asynchronous communications and remote code execution. One of the most interesting aspect of $_{d}$SL is how $_{d}$SL programs are designed. Indeed, $_{d}$SL programs are designed as if all input and output were centralised (i.e. the controller is composed by only one site). It is during the compilation that the distribution of the code is performed among the different sites.

But, as $_{d}$SL is a language for controllers of industrial systems, it may be used in critical environments. In this case, one must be confident with the program that will be used. For this, we could perform automatic verification on $_{d}$SL programs. Unfortunately, it is difficult to perform automatic verification of real cases as this corresponds to manipulations of complicated algorithms and huge data-structures.

After presenting $_{d}$SL and its semantics, we presented two techniques that should help to perform verification of $_{d}$SL programs. The first technique was using an interesting theoritical result on $_{d}$SL. Indeed, we showed that we did not have to perform verification on all possible distributions of a $_{d}$SL program. *Simulation* relations exist between the semantics of $_{d}$SL using different distributions. We had that, when collapsing two sites, the new distributions had less *behavior* than the original one. We can use this simulation relation to help the verification. Moreover, using the relation between distributions, we can build a lattice with the semantics using different distributions. This lattice is useful in the verification as it gives a clear view of what can be verified.

The second technique is *slicing*. We applied slicing techniques on $_{d}$SL language. We started from a usual formalism for slicing and applied it to $_{d}$SL. We discovered that usual techniques suffered from over-conservative slicing and we presented a more refined technique to perform slicing. After presenting some experiments, we discovered that slicing was useful in $_{d}$SL. Indeed, when concentrating on a given output variable, we saw that we could remove useless code controlling other output variables.

Future works

This papers gives two interesting techniques to start the verification of $_{d}$SL. It now remains to perform the verification in itself. For this, we should study the existing verification techniques an try to apply them to $_{d}$SL. A case study on the verification of $_{d}$SL using the model checker Spin has already been performed [DMM03].

We presented a simulation relation between semantics of $_{\mathrm{d}}$SL using different distributions. Maybe we could find other relations between distributions of $_{\mathrm{d}}$SL. Indeed, it should be interesting to derive some properties on a given distribution of a $_{\mathrm{d}}$SL program from the verification of a less distributed program.

The slicing technique we presented in this paper is still over-conservative. Indeed, expressions, and variable's value were not interpreted. We could try to refine slicing, considering paths that was used to include a given instruction in the slice. We could try to interpret intermediary instructions and try to find if this path is feasible. For example, we could try to determine if an assignement may or not trigger a when.

Bibliography

[Aub97] P. Aubry. *Mises en oeuvre distribuées de programmes synchrones (thèse).* Phd thesis, IFSIC, Rennes, France, October 1997.

[BB91] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *Proceedings of the IEEE*, volume 79, pages 1270–1282, 1991.

[BC85] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, 1985.

[Ber89] G. Berry. Real time programming: Special purpose or general purpose languages. Information Processing 89, G.X. Ritter (Ed.), Elsevier Science Publishers B.V., North-Holland, 1989. 11-18.

[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.

[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.

[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.

[DMM03] Bram DeWachter, Thierry Massart, and Cedric 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.

[Esk90] M. Rasit Eskicioglu. Design issues of process migration facilities in distributed systems. In *IEEE Computer Society Technical Committee on Operating Systems and Application Environments Newsletter*, volume 4, pages 3–13, 1990.

[Gir94] A. Girault. *Sur la Répartition de Programmes Synchrones.* Phd thesis, INPG, Grenoble, France, January 1994.

[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.

[HR92] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th international conference on Software engineering*, pages 392–411. ACM Press, 1992.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[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.

[KKP$^+$81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–218. ACM Press, 1981.

[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.

[Mas92] T. Massart. A calculus to define correct transformations of LOTOS specifications. In *Proceedings of the FORTE'91 conference*, pages 281–296, 1992.

[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.

[Mor99] René Morin. Decompositions of Asynchronous Systems. In *Proc. CONCUR'98, Springer Lect. Notes in Comp. Sci. 1466*, pages 549–565. Springer, 1999.

[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.

[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.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184. ACM Press, 1984.

[RHSR94] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 11–20, 1994.

[SEM03] Alin Stefănescu, Javier Esparza, and Anca Muscholl. Syntesis of Distributed Algorithm. In *To appear at 14th international conference on concurrency theory (CONCUR 2003)*, 2003.

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[WGMM04] Bram De Wachter, Alexandre Genon, Thierry Massart, and Cédric Meuter. dsl to design distributed industrial control systems and verify them with *spin*. *Formal Aspects of Computing - Special issue for AVOCS 2003.*, 2004. Submitted to FAC 2003: Formal Aspects of Computing. Special issue for AVOCS 2003.

Appendix A

Grammar

Remark *This grammar is a reduced version of the complete $_d$SL grammar, and to keep it compact, some rules may introduce ambiguity.*

Dsl_Program	⟶ `global_var` *dsl_gdecl_list* `end_var` *dsl_element_list init*
dsl_gdecl_list	⟶ *global_var_declaration dsl_decl_list* | *global_var_declaration*
dsl_element_list	⟶ *dsl_element dsl_element_list* | *dsl_element*
global_var_declaration	⟶ *id*: *var_kind* *var_type*;
dsl_element	⟶ *method* | *when* | *sequence*
init	⟶ `sequence init begin_var` *var_list* `end_var` *instruction_list* `end_sequence`
when	⟶ `when` *expr* `then` *instruction_list* `end_when`
method	⟶ `method` *id* `(` *param_list* `)` `begin_var` *var_list* `end_var` *instruction_list* `end_method`
sequence	⟶ `sequence` *id* `(` *param_list* `)` `begin_var` *id_list* `end_var` *instruction_list* `end_sequence`
param_list	⟶ *id* `:` *var_type* | *id* `:` *var_type* `,` *var_list*
var_list	⟶ *id* `:` *var_type*; | *id* `:` *var_type* `;` *var_list*
var_type	⟶ `int` | `bool`
var_kind	⟶ `internal` | `input` | `output`
instruction_list	⟶ *instruction instruction_list* | *instruction*
instruction	⟶ *if* | *while* | *assign* `;` | *synchronous_method_call* `;` | *asynchronous_method_call* `;` | *sequence_call* `;`
if	⟶ `if` *expr* `then` *instruction_list* `else` *instruction_list* `end_if`
while	⟶ `while` *expr* `do` *instruction_list* `end_while`
assign	⟶ *id* `:=` *expr*

synchronous_call ⟶ *id* `(` *id_list* `)`

asynchronous_call ⟶ `launch` *synchronous_method_call*

sequence_call ⟶ `launch` *id* `(` *id_list* `)`

id_list ⟶ *id* | *id* `,` *id_list*

expr ⟶ *expr bin_op expr* | *un_op expr* | `(` *expr* `)` | *id* | *cst*

bin_op ⟶ `+` | `-` | `*` | `/` | `<` | `>` | `<>` | `<=` | `>=` | `and` | `or`

un_op ⟶ `-` | `not`

Appendix B

Translation from $_{\mathrm{d}}$SL to $_{\mathrm{d}}\mathrm{SL}_{\Diamond}$

B.1 Suppressing sequences

We now explain how to transform a `SEQUENCE` into a list of `WHEN`'s and `METHOD`'s. The problem comes from the fact that a `SEQUENCE`'s execution may be distributed among various sites. When a given site executing a `SEQUENCE` cannot execute the next instruction (because it is located on another site), it sends an *hand-over* message to the site owning this instruction. This site will continue the execution of this `SEQUENCE` when it will receive this message. We can simulate this mechanism as follows. Let's suppose we want to simulate a `SEQUENCE` s defined by

$$\textit{sequence\_decl} \longrightarrow \texttt{SEQUENCE}\ \textit{id}\ \texttt{(}\ \textit{param\_list}\ \texttt{)}\ \texttt{begin\_var}\ \textit{decl\_list}\ \texttt{end\_var}\ \textit{instruction\_list}\ \texttt{END\_SEQUENCE}$$

Here, *param_list* is the list of parameters of this `SEQUENCE`, and *decl_list* is the list of local variables of this `SEQUENCE`.

For each instruction i contained in a sequence, we define a program counter variable PC_i controlling the execution of i. This boolean variable will be set to $\top$ followed by $\bot$ by i's predecessor to trigger the execution of i through a `WHEN` conditionned on PC_i. However, since `SEQUENCE`s may be distributed on different sites, the assignment to PC_i may possibly require an asynchronous execution. This is the case when the predecessor of i and i are in different partitions of the distribution. Furthermore, for each parameter or local variable v, we have a global variable v_i, which is the copy of v needed to execute i. We set these variables as global because $_{\mathrm{d}}\mathrm{SL}_{\Diamond}$ does not allow local variables nor parameters. Then, i is modified to access v_i instead of v. This can be accomplished by simply making a string substitution in the program code.

The translation of each sequential instruction is straightforward using the previous definitions. For each instruction i but the first one in a sequence S, we create a method $Trigger_i(\textit{param\_list}, \textit{decl\_list})$ defined as follows

```
METHOD Trigger_i(param_list,decl_list)

   Vi  := V; ∀ V ∈ param_list ∪ decl_list
   PCi := true;
   PCi := false;
END_METHOD
```

This method initialises each local copy of the variables and parameters to the values transmitted before triggering the `WHEN` that will execute the instruction.

Depending on the nature of i, we also define the following `WHEN` W

- If i is of the form `IF` e `THEN` i_1 `ELSE` i_2 `END_IF` i', then W is defined as

```
WHEN PCi THEN
 IF e THEN trigger(i1); ELSE trigger(i2); END_IF
END_WHEN
```

where `trigger`(j) is defined as $\begin{cases} \texttt{Trigger\_j(param}_i\texttt{, local}_i\texttt{);} & \text{if } \textsc{loc}(i) = \textsc{loc}(j) \\ \texttt{LAUNCH Trigger\_j(param}_i\texttt{, local}_i\texttt{);} & \text{if } \textsc{loc}(i) \neq \textsc{loc}(j) \\ \epsilon & \text{if } j = \epsilon \end{cases}$

Here, param_i(resp. local_i) is the list of parameter variables (resp. local variables) for instruction i.

- If i is of the form `WHILE` e `DO` i_1 `END_WHILE` i', then W is defined as

```
WHEN PCi THEN
 IF e THEN trigger(i1); ELSE trigger(i'); END_IF
END_WHEN
```

Where `trigger`(j) is defined as before.

- If i is an assignment, synchronous, or asynchronous call then, if $Succ(S, i) = \{j\}$, we define a `WHEN` W as :

```
WHEN PCi THEN i; trigger(j); END_WHEN
```

Note that j can be equal to ϵ, i.e. $Succ(S, i)$ is empty.

B.2 Single execution of SEQUENCE's

During the execution of the original $_{\rm d}$SL program, any sequence can only be executed at most once simultaneously. (i.e. the code is not reentrant). As the `SEQUENCE`'s are now splitted in multiple `WHEN`'s and `METHOD`'s, we must ensure that we won't have two `LAUNCH`'es of the same `SEQUENCE` executed at the same time. For this purpose, we introduce the following `WHEN` :

```
WHEN DONE(i_1) AND ... AND DONE(i_n) THEN PC_f := FALSE;  END_WHEN
```

Where *instruction_list.last* $= \{i_1, \ldots i_n\}$, $f =$ *instruction_list.first*, and `DONE(i)` is defined as follows,

$\begin{cases} \texttt{NOT PC\_i} & \text{if } \textsc{loc}(i) = \textsc{loc}(f) \\ \texttt{NOT} \sim\texttt{PC\_i} & \text{if } \textsc{loc}(i) \neq \textsc{loc}(f) \end{cases}$

This `WHEN` simply allows a new execution when the last one is over. It knows that this execution is over when a PC variable of one of the last instructions switches to `false`. The next section explains how to translate `LAUNCH`'es in this context.

B.3 Removing LAUNCHes

There is a close connection between the LAUNCH and the $\sim$ operator. The first one operates on some code, asking for the asynchronous execution of this code, while the latter is used to asynchronously monitor a global variable on a distant site. We now explain how to translate a LAUNCH into $_{\rm d}$SL$_\Diamond$ using $\sim$. Remark that this translation needs a variable per launched METHOD or SEQUENCE and per LAUNCH instruction. Indeed, since the assignment to this variable may only be done on the site "owning this call", each LAUNCH should have its own copy.

Formally, for each instruction i of the form LAUNCH *id* ($e_1, e_2, ..., e_k$) :

1. We introduce a new variable `launched_id_i`.

2. We add a new WHEN w. Depending on the type of *id* (i.e. METHOD or SEQUENCE), w will be different.

   - If we have a SEQUENCE s defined as :
     SEQUENCE *id* (*param_list*) begin_var *decl_list* end_var *instruction_list* END_SEQUENCE.
     Then we define w as follows :

     ```
     WHEN ~launched_id_i THEN
       IF NOT PC_f THEN
         Vf  := ~Vi;      ∀ V ∈ param_list, decl_list
         PC_f := true;
       END_IF
     END_WHEN
     ```

     where f = *instruction_list.first* is the first instruction of the SEQUENCE. If the sequence is not already launched, this WHEN initialises the local copy of the SEQUENCE's variables and parameters, then it triggers the WHEN corresponding to the first instruction

   - If we have a METHOD m defined as :
     METHOD *id* (*param_decl*) begin_var *var_decl* end_var *instruction_list* END_METHOD, then w is defined as follows :

     ```
     WHEN ~launched_id_i THEN
        Vf  := ~Vi; ∀ V ∈ param_list, decl_list
         instruction_list
     END_WHEN
     ```

3. we replace the LAUNCH instruction i with :

   ```
   Vi = ei;      ∀ Vi ∈ param_list
   launched_id_i := FALSE;
   launched_id_i := TRUE;
   ```

   As we will see later, the tilde messages are send using a FIFO channel. Thus, the `~Vi` copies of variable `Vi` for the site owning the METHOD/SEQUENCE will be updated before the `~launched_id_i` is set to true.

Appendix C

Notations

This section gives a summary of the most important notations used in chapter 4 on slicing.

Symbol	
P	Denotes a well-formed $_{d}$SL program.
CFG $C = (V_C, E_C)$	The Statement Level Control flow graph, defined page 32.
SDG $G = (V, E)$	The System Dependence Graph. This is the main control structure used in slicing and it is defined page 33.
v	Denotes a vertex in V or V_C (depending on the context).
ρ	Denotes a path in the CFG or the SDG (depending on the context).
$x \in Var_A(P)$	Denotes a variable in P.
V_a	Nodes in the CFG and the SDG that correspond to assignement
V_{ep}	Nodes in the CFG and the SDG that correspond to entry or exit points (`WHEN`, `METHOD`, or `SEQUENCE`).
e	Denotes an entry point.
$V_\top \subseteq V_{ep}$	The subset of entry points in V_{ep}.
$V_\bot \subseteq V_{ep}$	The subset of exit points in V_{ep}.
$V_{meth} \subseteq V_{ep}$	The subset of `METHOD`s in V_{ep}.
$V_{when} \subseteq V_{ep}$	The subset of `WHEN`s in V_{ep}.
$w \in V_{when}$	Denotes a `WHEN`.
$V_{seq} \subseteq V_{ep}$	The subset of `SEQUENCE`s in V_{ep}.
V_{call}	Nodes in the CFG and the SDG that correspond to calls.
$c \in V_{call}$	Denotes a call.
$V_{scall} \subseteq V_{call}$	The subset of synchronous calls of V_{call}.
$V_{acall} \subseteq V_{call}$	The subset of asynchronous calls of V_{call}.
V_{if}	Nodes in the CFG and the SDG that correspond to `if` statements.
V_{while}	Nodes in the CFG and the SDG that correspond to `while` statements.
$V_{\mathrm{n}} \equiv V_C$	The set of nodes in the SDG that correspond to real nodes (i.e. nodes that also appear in the CFG).

$V_{\text{ao}} \subseteq V$	The set of `actual out vertices` used for inter procedural slicing.
$ao(\in V_{\text{ao}})$	Denotes an `actual out vertex`.
$V_{\text{ai}} \subseteq V$	The set of `actual in vertices` used for inter procedural slicing.
$ai(\in V_{\text{ai}})$	Denotes an `actual in vertex`.
$V_{\text{fo}} \subseteq V$	The set of `formal out vertices` used for inter procedural slicing.
$fo(\in V_{\text{fo}})$	Denotes a `formal out vertex`.
$V_{\text{fi}} \subseteq V$	The set of `formal in vertices` used for inter procedural slicing.
$fi(\in V_{\text{fi}})$	Denotes a `formal in vertex`.
$E_{\text{ct}} \subseteq E$	The subset of control edges of E.
$E_{\text{f}} \subseteq E$	The subset of flow edges of E.
$E_{\text{fs}} \subseteq E_{\text{f}}$	The subset of synchronous flow edges of E_{f}.
$E_{\text{fa}} \subseteq E_{\text{f}}$	The subset of asynchronous flow edges of E_{f}.
$E_{\text{call}} \subseteq E$	The subset of call edges of E.
$E_{\text{cs}} \subseteq E_{\text{call}}$	The subset of synchronous call edges of E_{call}.
$E_{\text{ca}} \subseteq E_{\text{call}}$	The subset of asynchronous call edges of E_{call}.
$E_{\text{t}} \subseteq E$	The subset of trigger edges of E.
$E_{\text{ts}} \subseteq E_{\text{t}}$	The subset of synchronous trigger edges of E_{t}.
$E_{\text{ta}} \subseteq E_{\text{t}}$	The subset of synchronous trigger edges of E_{t}.
$E_{\text{par}} \subseteq E$	The subset of parameter edges of E.
$E_{\text{pi}} \subseteq E_{\text{par}}$	The subset of parameter in edges of E_{par}.
$E_{\text{po}} \subseteq E_{\text{par}}$	The subset of parameter out edges of E_{par}.
$R \subseteq V_{\text{ao}}$	The set of real `actual out` vertices.
$S \subseteq 2^V \times 2^V$	The set of summary edges.
cl_ρ	Call level function, defined page 49.
ce_ρ	Corresponding entry function, defined page 49.
cx_ρ	Corresponding exit function, defined page 49.

Appendix D

Algorithm to compute summary edges

Procedure `ComputeSumEdges(`G`)` - Algorithm for computing summary edges (revisited)

Data : $G = (V, E)$, the dependence graph.
Result: S, the set of summary edges, and R the set of real `actual out` vertices.
begin
  $E_{\text{pe}} \leftarrow \{(v, v) | v \in V_{\text{fo}}\}$, $E_{\text{w}} \leftarrow \{(v, v) | v \in V_{\text{fo}}\}$;
  $R \leftarrow \emptyset$;
  while $E_w \neq \emptyset$ **do**
    Let $e = (v, v') \in E_{\text{w}}$;
    $E_{\text{w}} \leftarrow E_{\text{w}} \setminus \{e\}$;
    if $v \in V_{fi} \wedge real(e)$ **then**
      /*`We have a path from a formal in vertex to a formal out vertex` */
      for *each* $ai \in V_{ai}$, $ao \in V_{ao}$ *such that*
      $\text{NORM}(ao) = \text{NORM}(ai) \ \wedge \ (ai, v) \in E_{pi} \ \wedge \ (v', ao) \in E_{po}$ **do**
        $S \leftarrow S \cup \{(ai, ao)\}$;
        for *each* $(ao, v'') \in E_{pe}$ *such that* $(ai, v'') \notin E_{pe} \vee \neg real((ai, v''))$ **do**
          $E_{\text{pe}} \leftarrow E_{\text{pe}} \cup \{(ai, v'')\}$, $E_{\text{w}} \leftarrow E_{\text{w}} \cup \{(ai, v'')\}$;
          mark (ai, v'') as *real*;
        end
    else
      if $v \in V_{ep} \wedge real(e)$ **then**
        for *each* $ao \in V_{ao}$ *such that* $(v', ao) \in E_{po}$ **do**
          $R \leftarrow R \cup \{ao\}$;
          for *each* $(ao, v'') \in E_{pe}$ *such that* $(\text{NORM}(ao), v'') \notin E_{pe} \vee \neg real((\text{NORM}(ao), v''))$
          do
            $E_{\text{pe}} \leftarrow E_{\text{pe}} \cup \{(\text{NORM}(ao), v'')\}$, $E_{\text{w}} \leftarrow E_{\text{w}} \cup \{(\text{NORM}(ao), v'')\}$;
            mark $(\text{NORM}(ao), v'')$ as *real*;
          end
      else
        for *each* $e' = (v'', v) \in E_{ct} \cup E_{fs} \cup S$ *such that* $\neg real(e')$ **do**
          if $e' \notin E_{ct} \vee v \notin V_{ao} \setminus R$ **then**
            if *real*(e) **then**
              mark e' as real;
              $E_{\text{pe}} \leftarrow E_{\text{pe}} \cup \{e'\}$, $E_{\text{w}} \leftarrow E_{\text{w}} \cup \{e'\}$;
            else
              if $\bigvee \begin{pmatrix} v'' \in V_n \wedge \text{DEF}(v'') = \text{USE}(v') \\ (v'', v) \in S \wedge \text{DEF}(v) = \text{USE}(v') \\ v \in R \wedge \text{DEF}(v) = \text{USE}(v') \end{pmatrix}$ **then**
                mark e' as real;
                $E_{\text{pe}} \leftarrow E_{\text{pe}} \cup \{e'\}$, $E_{\text{w}} \leftarrow E_{\text{w}} \cup \{e'\}$;
              else
                if $e' \notin E_{pe}$ **then**
                  $E_{\text{pe}} \leftarrow E_{\text{pe}} \cup \{e'\}$, $E_{\text{w}} \leftarrow E_{\text{w}} \cup \{e'\}$;
                end
              end
            end
          end
        end
  return (S, R)
end