Under consideration for publication in Formal Aspects of Computing

The formal design of distributed controllers with $_{\rm d}$SL and Spin

Bram De Wachter[1], Alexandre Genon[1], Thierry Massart and Cédric Meuter[1]

{*bdewacht, agenon, tmassart, cmeuter*}*@ulb.ac.be*,
Université Libre de Bruxelles,
Computer Science Department,
ULB CP212, boulevard du Triomphe, 1050 Bruxelles,
Belgium

Abstract. We study the formal verification of programs written in $_{\rm d}$SL, an extension of the standard ST language used to program industrial controllers. It proposes a trade off between industrial and formal verification worlds. The main advantage of $_{\rm d}$SL is to provide a transparent code distribution through low level communication mechanisms. The behavior of the synthesized distributed system can therefore be formally modeled, easily monitored and formally verified. The verification of a $_{\rm d}$SL program, realized with the *Spin* tool, is eased by the definition of a lattice of models linked with a simulation relation preserving next-free LTL formulae. We show that, although $_{\rm d}$SL is an industrial programming language, it gives the possibility to verify systems designed with it. We illustrate the benefit of our approach with a simple control system of two canal locks.

Keywords: Industrial process control, transparent code distribution, verification, *Spin*

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 exist, (CORBA, DCOM, EJB,...). They handle the communication aspects and allow the programmer to concentrate on the functional aspects of the system. Unfortunately, these solutions are generally quite heavy, and the communications aspects are not formally specified. These systems are also difficult to monitor because of their complexity.
In a previous work we introduced a solution to this problem in the form of the $_{\rm d}$SL language [DeWMM03] which is a simple, imperative and event driven language. It is specifically designed to program distributed industrial control systems and is an extension of the standard ST language [BMS97] widely used in the

[1] Work supported by the *Region de Bruxelles Capitale*, grant no. RBC-BR 227/3298.
Correspondence and offprint requests to: B. De Wachter, A. Genon, T. Massart and C. Meuter

industry to program such controllers. The main advantage of dSL is at the programming level: it provides a transparent code distribution using low level communication mechanisms. Therefore, most of the time, the dSL programmer can ignore all the communication aspects between controllers of the distributed system. This automatic distribution mechanism is however not the subject of this paper. More details on that can be found in [DeWMM03, DeWGM05].
In this paper, we study the problem of formally verifying systems designed with dSL. In order to do so, we first give the formal semantics of dSL in terms of state machines communicating through FIFO channels. We show that although dSL is a high-level automatically distributed programming language, the behavior of the synthesized system can still be formally modeled and easily monitored - which is an important concern for these kind of systems. However, because of the distributed nature of dSL, one program may lead to several different semantics, depending on the physical environment in which it will execute. Fortunately, the main theoretical result of this paper shows the existence of a relation between all these semantics which can be exploited during the verification.
On the practical side, we show how to translate dSL into *Promela*, the language used by the *Spin* model checker, using this formal semantics. As a case study, we illustrate our approach on the design of a control system of two canal locks.
The paper is organized as follows: First, in section 2, we detail related proposals and justify the relevance of our solution. Then, in section 3, we give an informal introduction of the dSL language. Next, in section 4, we present the syntax and the formal semantics of a subset of the dSL language. In section 5, we discuss the relation between the different distributions of a given dSL program and how it can be used during the verification phase. Section 6 presents the translation of dSL to *Promela*. The case study, a locks controller designed in dSL, is presented in section 7. Finally, some concluding remarks are given in section 8.

2. Motivations and other approaches

The problem of distributing applications that control reactive systems has been studied for many years now and several interesting observations made on these studies shaped the design of dSL. We comment in more detail works on process algebra, Synchronous Languages, and higher level frameworks such as Unity. The choice of the distributed execution environment used by dSL is motivated by a discussion on shared memory systems and thread migration.

Other approaches In the world of process algebra, the problem of automated distribution is defined as a correctness preserving transformation of a *centralized* 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 labeled transition systems ([CMT99, Mor99, SEM03]). These works solved part of the problem. However, contrary to programming languages, the notion of variables does not exist in these formalisms and other solutions had therefore to be found.
In a higher level framework, Chandy and Misra have proposed the Unity approach [KMC88] to model and design asynchronous or synchronous parallel programs. Let us recall that the principle in Unity, similar to the one proposed by the B method [Abr96], is to use a design modeling language together with a proof system to provide, through several design decisions, correct parallel programs. Central in this framework is the separation between the concern of program development and the physical architecture on which it is implemented. The program development phase provides the specification of the Unity program in itself using guarded, multiple assignment statements. During the second phase, starting from a Unity program, a *mapping* to an architecture is used to describe a possible implementation. Various possibilities are proposed to implement a Unity program on a distributed architecture. The program variables can be seen as shared variables or communication can be done through FIFO channels. In both cases, a protocol must be explicitly given to preserve the data integrity or synchronize the execution flow. This implies that, each time the target architecture is modified (e.g. adding or removing processor(s), moving variable(s), refining the distribution, . . .), the Unity program has to be modified at the communication level, to fit this new distribution. Therefore, the Unity program must be designed with a desired mapping in mind, which has a negative impact on the reconfiguration flexibility of applications and the transparency of the physical distribution.
On the programming language side, the most relevant works on automated distribution of reactive systems have been done in the domain of synchronous languages such as Esterel [BG92], Lustre [CPHP87] and Signal [LGLL91], which answered questions on how to specify controllers in a natural and semantically well defined

way. Unfortunately, because the semantics must be preserved, the distribution of synchronous languages suffer from a performance problem which, in practice, may not be acceptable. Indeed, the synchronous programming scheme found in these languages supposes that time is defined as a sequence of *instants* which are common to all parallel processes contained in the specification. Although this allows the use of synchronous broadcast [Ber89, BB91] resulting in sequential code, a strong synchronization scheme must be used to preserve these instants when such programs are distributed [Gir94, Aub97]. This strong synchronization has several 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. We believe therefore 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 practice, the strong synchronization of all processes is rarely needed and must therefore not be used as a default.
These observations motivate the makeup of $_d$SL. At the design level, Unity seems more sound than what is proposed with the direct use of programming languages (among which $_d$SL), since a Unity design goes through the correctness proof for the design before doing the implementation. Unfortunately, the industrial control world has not yet integrated this more formal approach to design real systems, and still uses more specialized programming languages defined as industrial standards. The philosophy of $_d$SL goes therefore in the other direction: it proposes a language close to the used standards and provides a framework to formally verify the designed systems. With the drawbacks of synchronous languages in mind, $_d$SL rejects the synchronous product [Mil81] used in synchronous languages at the detriment of indeterminism, and adopts semantics based on asynchronous composition of local instantaneous code and global distributed code. We shortly comment how the execution environment handles both codes.

Execution Environment For the distributed code, several models of execution are proposed in the literature. These models can be divided into two sets based on the way they achieve data locality : either move the data to the executing processor, or move the execution to the processor which has the data in memory. Many systems based on the first solution, such as Distributed Shared Memory systems [NL91], have been studied. These systems, despite offering a transparent distributed environment, suffer from undesirable border effects which are generally problematic in an industrial environment. First, the need to replicate data in these systems [HHG99], may result in unstable performance as observed in [NL91]. Secondly, since the data moves around, the supervision of such systems and its error-recovery - both indispensable features in industrial applications - may become too complicated [MP97] on $_d$SL's target hardware. For these reasons, $_d$SL uses the second solution, a concept known as process or thread migration [Esk90] where a thread of execution is halted on one site, its context (local variables and program counter) is sent to another site, where its context is restored and execution continues. Thread migration is known to enable dynamic load distribution, fault tolerance, eased system administration, data access locality and mobile computing [JC02]. However, in our system, all instructions and global variables are statically assigned to the participating sites and thread migration, determined at compile time, is used to obtain data access locality. This has three benefits: (1) following the state of the system is very easy, (2) all communication and synchronization messages can statically be calculated, resulting in a predictable execution, (3) the communication between processes can, in practice, be reduced to very low level mechanisms. However, we lose the benefits of dynamic load balancing and fault tolerance[2] since the migration policy used in $_d$SL is static.
The local code uses an event driven scheme and, for a given component, must be able to run without any synchronization that would make it wait on other components. This asynchronous composition has the advantage that the failure of one site does not introduce deadlocks in *atomic* code on other sites. The *sequential* code, on the other hand, can be executed in a totally distributed and cooperative manner. These assumptions, of course, imply some restrictions on code and have consequences on the way data values are transmitted between distributed processes.

The shape of $_d$SL can thus be synthesized as an imperative language using a hybrid execution scheme

[2] Fault tolerant $_d$SL systems can be obtained through the additional possibility to test failures in the communications and hardware (see "unknown values" in [DeWMM03]); but needs extra coding and the explicit use of redundancies.

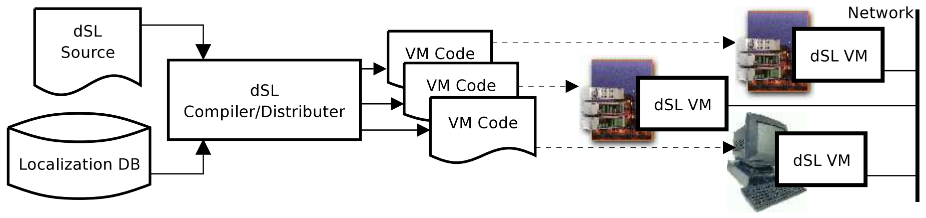

Fig. 1. $_{\mathrm{d}}$SL architecture

composed of two types of code : local or *atomic* instantaneous code, and distributed *sequential* code that executes using statically calculated thread migration.

3. Informal introduction to $_{\mathrm{d}}$SL

$_{\mathrm{d}}$SL [3] 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). $_{\mathrm{d}}$SL has kept the general way industrial controllers operate through an input sampling, treatment, writing cycle. However, in $_{\mathrm{d}}$SL each site executes such cycles asynchronously from the others. $_{\mathrm{d}}$SL is event driven, i.e., an event is specified by the change in some variable's value. $_{\mathrm{d}}$SL also offers limited Object-Oriented features. Variables and methods can be structured into objects, but we will not detail this aspect any further in this paper.
As illustrated in figure 1, a $_{\mathrm{d}}$SL program is written as if the entire environment can be accessed without the need for explicit communication or synchronization primitives (we shall see that some restrictions are imposed to apply this principle). The designer of such program must then provide a *localization table* to specify the physical localization (site) of each I/O variable. The $_{\mathrm{d}}$SL compiler then automatically distributes the code among the different execution sites. In this phase, the distributer optimizes the distributed code in order to minimize the expected number of exchanged messages during execution. We show in [DeWGM05] that the production of efficient code can be formulated as an optimization problem on weighted graphs, which is shown equivalent to the NP-Hard multiterminal cut problem [DJP$^+$94].
The distributed code is compiled to an assembler-like language. This assembler-like code is interpreted by a set of $_{\mathrm{d}}$SL *Virtual Machines* which are interconnected through a network and transmit data through FIFO queues. A $_{\mathrm{d}}$SL virtual machine is available for each type of execution site. Indeed, an execution site can either be a supervisor, i.e. a computer with a user interface, or a programmable controller connected to the industrial equipment to control through sensors and actuators. This equipment will be referred to as the *environment* in the rest of the paper.
The benefits of this approach are (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 synchronization schemes to handle particular tasks). Another advantage, as we will see later, is to be able to verify the correctness of a program, independently of the distribution of the variables among the different execution sites.

Atomic and sequential code As we have mentioned previously, the design of $_{\mathrm{d}}$SL has been dictated by the execution paradigm used in the world of industrial process control that requires an immediate reaction to events and their instantaneous treatments. In practice, by default, this forbids any hidden synchronization delaying execution and in particular synchronization which implies inter-site communications (through a relatively slow network). A clear way must therefore exist to express that inter-site synchronization is allowed. Hence $_{\mathrm{d}}$SL distinguishes between two types of code:

- *atomic* or *event-driven* code which must execute in an atomic manner, and can therefore not be distributed
- *non atomic* or *sequential* code which can execute in a distributed manner, using inter-site communications to synchronize or transfer values between sites.

[3] $_{\mathrm{d}}$SL 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

Event-driven code can be defined using the `WHEN` keyword. This implies that two variables appearing in the same `WHEN` must be localized on the same execution site. To relax this constraint, two mechanisms have been defined: (1) the `LAUNCH` instruction allowing to start *sequential* code, and (2) the "~" operator.
The principle of the tilde operator is to access a local variable, whose value is the last known value of a remote one. For every variable `x`, the dSL semantics imposes that each site i has also a variable `~x`. Lets denote by $\nu_i(\tilde{x})$ the value of the copy of `x` local to site i. Every time `x` is modified, each $\nu_i(\tilde{x})$ is updated asynchronously through a broadcast over the Fifo channels. Therefore, the code `y := ~x` where `y` is local to site i corresponds to `y :=` $\nu_i(\tilde{x})$, i.e. `y` receives some, maybe old value of the (remote) variable `x`.
This feature becomes useful when a program is not distributable. Indeed, one can use the local `~x` variable instead of requiring direct access to `x`, which constraints more the dSL program (see Example program). Using `~x`, the programmer now has a more flexible dSL program that may be further distributed. But *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.
On the other hand, *sequential* code can be defined using the `SEQUENCE` keyword. A `SEQUENCE` cannot have more than one instance executed simultaneously. A `SEQUENCE` can have *local* variables (i.e. which scopes are limited to the `SEQUENCE`). When needed, these variables can be transmitted from site to site to ensure a coherent execution of the `SEQUENCE`. Note that `SEQUENCE`s are exclusively called through the use of `LAUNCH`s.

Example program To illustrate the dSL concepts, let us give a simplified example of a program that receives input values through an input variable `temperature`, linked to a temperature sensor and switches on (off) a `heater` when the temperature is below 0 ^{o}C (above 20 ^{o}C). Two indicators in a control panel are used to monitor the status of the heater. The first one (`led`) indicates the state of the heater and the second one (`alarm`) indicates that a maintenance check is needed. For that purpose, a boolean variable `maintenance` is set when the heater has been turned on or off a thousand times. The program will be used in a physical configuration with 2 execution sites: (1) the control panel, where the output variables `led` and `alarm` will be located and (2) the heater where the output variable `heater` and the input variable `temperature` will be located. The location of the other variables are initially unspecified. However, at first sight, it would be impossible to have `maintenance`, `temperature` and `alarm` on the same execution site. Indeed, `WHEN`s `W2` and `W3` require `maintenance` to be on the heater site, whereas the following `WHEN` :

```
WHEN maintenance THEN
    alarm:=true;
END_WHEN
```

would require `maintenance` to be on the control panel site. Therefore, in order to make the distribution possible, we need to relax the constraints on one of those `WHEN`. In the example, presented in figure 2 we chose to use the local variable `~maintenance` in the condition of `WHEN W1`.

4. dSL's syntax and semantics

A dSL program contains 5 elements. (1) global variables declarations including all I/O variables (2) `METHOD` definitions (3) `WHEN` definitions (4) `SEQUENCE` definitions and (5) an initialization sequence. In order to keep the semantics simple, we present only a subset of the dSL language. In this subset, we make the following restrictions: (1) methods are supposed to be inlined, which implies that recursive calls are forbidden; (2) since no recursion is allowed, all variables are considered to be declared globally; (3) only boolean variables are considered; (4) `SEQUENCE`s and `LAUNCH`s are not considered. Indeed, although very convenient for the programmer, `SEQUENCE`s and `LAUNCH`s do not increase the expressiveness of the language and can be equivalently translated into code using only `WHEN`s and *tilde*, as explained in [Gen04]. In this paper, we only consider *well-formed* dSL programs where each used variable has been properly defined and each `WHEN` uses at least one global variable.
From now on, we will use the following notations:

- $Var(P)$ denotes the set of non tilded variables appearing in the program P. This set is partitioned into Var^{in}(P), Var^{out}(P) and Var^{τ}(P) which correspond respectively to the input, output and internal (i.e. non I/O) variables.

```
GLOBAL_VAR
  led         : BOOL;
  heater      : BOOL;
  alarm       : BOOL;
  temperature : BOOL;
  maintenance : BOOL;
  control     : 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 (* W1 *)
  alarm := TRUE;
END_WHEN
```

```
WHEN temperature < 0 THEN  (* W2 *)
(* Turn on the heater *)
   IF (NOT maintenance) THEN
      LAUNCH set_heater (TRUE);
   END_IF;
END_WHEN

WHEN temperature > 20 THEN  (* W3 *)
(* Turn off the heater *)
   IF (NOT maintenance) THEN
      LAUNCH set_heater (FALSE);
   END_IF;
END_WHEN

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

Fig. 2. simple $_\text{d}$SL program

- $Var(w)$ denotes the set of non tilded variables appearing in the `WHEN` w.
- $Var(e)$ returning the set of variables (*tilded or not*) appearing in expression e.
- $Whens(P)$ denotes the set of `WHEN`s appearing in the program P.
- $Cond(w)$ denotes the triggering condition of the `WHEN` w.
- $Body(w)$ denotes the body of the `WHEN` w.
- $\tilde{X}$ denotes the set of tilded variables corresponding to X, $\tilde{X} = \{\tilde{x} | x \in X\}$.

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

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

Preliminary definitions

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

Definition 1 (Labeled transition system). A labeled transition systems L is a tuple $(Q, q_0, \Sigma, \rightarrow)$ where :

- Q is a set of states,
- $q_0 \in Q$ 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 $q' = q_n$. Note that if $w = \epsilon$, we have $q = q_0 = q_n = q'$.

Additionally, the behavior of a $_{\text{d}}$SL program depends on the distribution of its variables. Therefore its semantics will be parametrized by a distribution of its variables. Intuitively, a distribution of a well-formed $_{\text{d}}$SL program is a partition of the set of its variables respecting the atomic constraints imposed by `WHEN`s. That is, if two variables appear un-tilded in the same `WHEN`, they must be localized on the same execution site. Let us define formally the notion of distribution.

Definition 2 (Distribution of a well-formed $_{\text{d}}$SL program). A distribution of a well-formed $_{\text{d}}$SL program P is a partition $D = \{V_1, V_2, ..., V_n\}$ of $Var(P)$ such that

$$\forall w \in Whens(P) \forall v, v' \in Var(w), \exists V \in D, (\{v, v'\} \subseteq V)$$

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

Moreover, given a well formed $_{\text{d}}$SL 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.

Definition 3 (Distributed execution context). Given a $_{\text{d}}$SL 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:

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

where each $W_i = \{w \in Whens(P) | Var(w) \subseteq V_i\}$. We will also need to divide W_i into the sets of whens conditioned on a specific variable : $W_{i/x} = \{w \in W_i \mid x \in Var(Cond(w))\}$
In the following, we will call each (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 $\forall i, j \in [1..n], (i \neq j) \implies (W_i \cap W_j = \emptyset)$ ♦

Finally, to abbreviate the semantics rules, we need to define some auxiliary functions.

Definition 4 (Input sampling, output writing). Given a set of variables V, we define $Sample(V)$, respectively $Write(V)$, as a list of instructions performing the input, respectively output, of the variables in V in the order of their declaration in the program text :

$$\begin{array}{rcl} Sample(\{I_1, .., I_k\}) & = & \texttt{INPUT}(\texttt{I}_1); ...; \texttt{INPUT}(\texttt{I}_\texttt{k}) \\ Write(\{O_1, .., O_{k'}\}) & = & \texttt{OUTPUT}(\texttt{O}_1); ...; \texttt{OUTPUT}(\texttt{O}_{\texttt{k}'}) \end{array}$$

♦

Definition 5 (Treatment of `WHEN`s). Given a set of `WHEN`s W, we define $Treat(W)$ as a list of instructions processing all `WHEN`s in W, in the order of appearance in the program text, as follows:

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

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

$$\omega_i = \texttt{IF}\ (Cond(w_i)\ \texttt{AND NOT}\ \mathit{old\_cond}_{w_i})\ \texttt{THEN}\ Body(w_i);\ \texttt{END\_IF}\ \mathit{old\_cond}_{w_i} := Cond(w_i);$$

The variables $\mathit{old\_cond}_{w_i}$ introduced here are fresh variables, added to $Var(P)$ and initially set to $\perp$. ♦

Structural operational semantics

Informally, the behavior of a $_{\text{d}}$SL program can be seen as the parallel composition of n processes, one for each site, communicating with the environment to control. Each process 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 *tilde* variables, which are the local value of the distant variables.
The execution of each process i is an infinite loop of cycles called *Input-Process-Output* cycle because it contains three phases. (1) *Input* : variables linked to inputs change their values according to the physical state of the device they are attached to, (2) *Process* : events are triggered, incoming messages are processed and (3) *Output* : variables linked to the outputs force the physical state of the devices they are connected to.
The formal structural operational semantics for a well formed $_{\text{d}}$SL program P, w.r.t. a distribution D is

given below as a labeled transition system which visible actions are updates to/from the environment of the I/O variables. Let us first define a global state of a $_{\text{d}}$SL program.

Definition 6 (Global state of a $_{\text{d}}$SL program). Given a distribution $D = \{V_1, V_2, ..., V_n\}$ of a well formed $_{\text{d}}$SL 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.
- $\nu_i : (V_i \cup \widetilde{Var(P)}) \mapsto \{\top, \bot\}$ is a valuation function for:
  - the global variables owned by process i.
  - the tilded (local) copies of all variables.
- $\phi_i \in ((Var(P) \times \{\top, \bot\}) \cup \{\diamond\})^*$ is the receiving communication channel of process i. Each message indicates the update of a variable. Additionally, $\diamond$ is used to enforce the end of the message treatment.

We will note $\mathcal{G}_D^P$ the set of global states of a $_{\text{d}}$SL program P; given a distribution D. ♦

We can now introduce the semantics rules which will provide the transition relation of the labeled transition systems. 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, unlike synchronous languages like Esterel or Lustre, $_{\text{d}}$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. Note that a message is also appended to 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]

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

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

The other rules described below are local rules; i.e. defining how a local process makes a move. The first of these local rules corresponds to the beginning of a new cycle. If, at a point in the execution, the workload of process i becomes empty (ε), then a new *Input-Process-Output* cycle is scheduled in the workload. Therefore, this rule dictates the cyclic behavior of each process. Note that, in order to model the non instantaneous behavior of the network, a $\diamond$ marker is inserted to delimit the messages (ψ_1) that will be treated during this cycle.

[Cycle start]

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

$$\phi_i' = \psi_1 \cdot \diamond \cdot \psi_2 \wedge \phi_i = \psi_1 \cdot \psi_2$$

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

[Input]

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

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 WHENs might be triggered. Thus, all WHENs that might be triggered by this change of valuation are considered before continuing the message treatment. Note that messages are of the form (x, v) where x is the updated variable and v is its value.

[Message treatment]

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

[End of message treatment]

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

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. The $\diamond$ marker, inserted when the cycle start start rule was fired, assures that new messages received during this cycle are not treated.
Note also that these rules can be modified to handle one FIFO channel between each couple of processes. We showed in [Gen04] that this does not change any result presented in this paper. For simplicity, we present here our semantics with only one channel for each process. However, the translation and the case study in section 6 is performed using the full semantics with a FIFO channel between each couple of processes.

The next rule corresponds to the treatment of an assignment in the workload. Besides the usual effect of the assignment (the local valuation is updated), the new value needs to be broadcast and the WHENs that may be triggered by this assignment need to be scheduled for treatment.

[Assignment]

$$(E_D^P)_i \vdash (x := e; \omega_i, \nu_i, \phi_i) \xrightarrow{\tau} (\texttt{BCAST(x)};Treat(W_{i/x}); \omega_i, \nu_i[x \mapsto \nu_i(e)], \phi_i)$$

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 evaluates 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 labeled by the output variable and its new value and removing this abstract instruction from the workload.

[Output]

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

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

Definition 7 (Distributed semantics of a $_{\text{d}}$SL program). Given a well-formed $_{\text{d}}$SL 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 labeled transition system $(\mathcal{G}_D^P, G_D^0, (Var(P) \times \{!, ?\} \times \{\top, \bot\}), \rightarrow)$ where:

- $G_D^0 = ((\omega_1, \nu_1, \phi_1), ..., (\omega_n, \nu_n, \phi_n))$ where $\forall i \in [1..n] : \omega_i = \varepsilon$, $\phi_i = \varepsilon$, $\forall x \in (V_i \cup \widetilde{Var(P)}) : \nu_i(x) = \bot$
- $\rightarrow$ is such that $(G, a, G') \in\rightarrow$ if and only if $E_D^P \vdash G \xrightarrow{a} G'$ can be derived from one of the structural operational semantics rules given previously.

♦

5. Hierarchy of distributions and semantics

In the previous section, we have seen that the semantics of a $_{\rm d}$SL program P is parameterized by a distribution of its variables. Therefore, if we want to formally verify P independently of its variables localisation, we should be verifying it w.r.t. every possible distribution. This is however clearly not acceptable. We will see in this section that fortunately, for a given $_{\rm d}$SL program P, there exists a lattice of *allowed* distributions. For two distributions D_1 and D_2 related by this lattice and with D_1 more distributed than D_2, we will show that the set of possible behaviors of D_2 is included in the set of possible behaviors of D_1. Hence, every valid trace property ϕ on P w.r.t D_1 is also valid on P for D_2. In particular if ϕ is valid for the *maximal distribution*, then it holds for any distribution. The contraposition of this results states that if a property ϕ does not hold on P for D_2, it does not hold either on P for D_1.
A case study, presented in the next sections, illustrates how these results can be used to verify completely or partially some program, depending on the computer resources available to verify it.

Definition 8 (Distribution refinement). Let $D = \{V_1, V_2, ..., V_k\}$ and $D' = \{V'_1, V'_2, ..., V'_l\}$ be two distributions of a $_{\rm d}$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 $_{\rm d}$SL program, which is the most refined distribution.

Definition 9 (Maximal distribution of a well-formed $_{\rm d}$SL program). The maximal distribution of a well-formed $_{\rm d}$SL program P is the distribution D_{max} such that

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

♦

Note that D_{max} is unique. Indeed, Suppose $D_1 \neq D_2$ both maximal, then there exist two global variables x, y with $\{x, y\} \subseteq V$ for some partition V of D_1 and $x \in V', y \in V''$ for some partitions V', V'' of D_2 with $V' \neq V''$. Therefore, by definition 2, it can be seen that V can be split into $V \cap V'$ and $V \cap V''$. By definitions 9 and 8, D_1 is therefore not maximal. Also note that two `WHEN`s are not necessarily on different sites in D_{max}, this is for example the case if they both use the same global variable.
Now we have to prove that this distribution induces a semantics that includes all possible behavior. For that, we use the notion of *weak simulation*. Let us recall this definition.

Definition 10 (Weak simulation relation). Given two labeled transition systems $L_1 = (Q_1, q_1^0, \Sigma, \rightarrow_1)$, $L_2 = (Q_2, q_2^0, \Sigma, \rightarrow_2)$. A binary relation $\mathcal{R} \subseteq Q_1 \times Q_2$ is a weak simulation relation for L_1 and L_2 if and only if for all $q_1 \in Q_1, q_2 \in Q_2, a \in \Sigma \cup \{\tau\}$, if $(q_1, q_2) \in \mathcal{R}$ then

$$\forall q'_1 : \left(q_1 \xrightarrow{a} q'_1\right) \implies \left(\exists q'_2 : q_2 \xrightarrow{\widehat{a}} q'_2 \wedge ((q'_1, q'_2) \in \mathcal{R}\right)$$

where

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

♦

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 $(q_1^0, q_2^0) \in \mathcal{R}$.
We can now prove that if a distribution D' of P refines distribution D of P $(D \preceq D')$, then $[\![P]\!]_D$ can be simulated by $[\![P]\!]_{D'}$ $([\![P]\!]_D \lesssim [\![P]\!]_{D'})$.

Theorem 1 (Simulation). Given a well-formed $_d$SL program P, 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. The proof can be found in appendix A. ■

Intuitively, this means that every step that can perform P with distribution D can be simulated by P using a more refined distribution D'. Theorem 1 implies the following well known results.

Corollary 1 (Ltl properties). Given a well-formed $_d$SL program P, let $D = \{V_1,\ V_2, ..., V_n\}$ and $D' = \{V_1', V_2', ..., V_l'\}$ be two distributions of P. If D' refines D then for every next free Ltl property Φ :

$$([\![P]\!]_{D'} \models \Phi) \Longrightarrow ([\![P]\!]_D \models \Phi)$$

■

This corollary and its contraposition are also useful in the verification phase.

Corollary 2 (Distribution lattice). Given a well-formed $_d$SL program P. Let $D_{min} = \{Var(P)\}$. We have that $\langle\ \{[\![P]\!]_D | D \in \mathcal{D}_P\}\ ,\ \lesssim\ \rangle$ is a lattice with respectively, $[\![P]\!]_{D_{min}}$ and $[\![P]\!]_{D_{max}}$ as minimal and maximal elements. ■

Therefore, a program P is safe for any distribution D if it is safe for the maximal distribution D_{max}. On the other hand, if P is shown not safe for some distribution D, then it is also not safe for any more refined distribution D'. This can be used in two ways to tackle the validation of a program P. First, one can use model-checking. In this case, considering P w.r.t its maximal distribution D_{max} is enough to ensure the validity of P. On the other hand, on can use testing to try and find errors. In such case, one can start with the centralized distribution D_{min}. Indeed, if an error is detected in this distribution, then it will be also be present in any other distributions.

6. $_d$SL to *Promela*

Aiming at the automatic verification of $_d$SL, we now briefly describe how to translate a $_d$SL program into *Promela*, the input language for the *Spin* model checker. The specification language *Promela* is much like any common imperative language (including global/local variables, `if`, `do` control flow and common expression evaluation) with a syntax close to the C programming language. The primary use of *Promela* is in formal specifications, it has not been designed for programming purposes. For example, it is enriched with non determinism inspired by Dijkstra's guarded command language. Moreover, *Promela* allows the dynamic creation of processes and has a rich set of primitives for interprocess communication, such as shared variables and Fifo queues. The *Promela* language has been designed with process specifications in mind, and computation should be abstracted before writing a specification in *Promela*. More details about this language can be found in [Hol03]. The semantics of $_d$SL allows an almost immediate translation of a given $_d$SL program into *Promela*. Remark, however, that this translation depends on the distribution used to run this program, meaning that the *Promela* specification will change every time another distribution is used. In this section, we first give the restrictions imposed by the use of *Promela*. Next, we describe how to efficiently model the environment. Finally, we detail the translation of the code for the different $_d$SL sites into *Promela*. A dedicated tool performs this translation automatically, except for the environment which must still be specified manually. This translation will be used to perform verification on the case study of section 7.

Restrictions Translating full $_d$SL into *Promela* is difficult for two reasons. First of all, recursive functions are difficult to translate into *Promela*. However, recursion is generally not a desired feature for industrial controllers. The second difficulty comes from the restrictive finite state space representation used by the *Spin* model checker, which imposes a bound on the sizes of the communication channels. Static analysis techniques can be used to detect problematic systems, where no such bound exists [LMW04].
The constructs omitted in the semantics description (`LAUNCH` and `SEQUENCE`) can easily be translated, either by replacing them using the constructs that are included in the semantics (a general description is given in [Gen04]), or by direct translation into *Promela*. The first solution is considered here.

Modeling the environment We have several solutions to model the environment of a $_{\rm d}$SL program in *Promela*. We could consider the environment as an individual process that reacts on outputs and continually changes the inputs of the $_{\rm d}$SL program. But this approach is quite inefficient. Indeed, consider the behavior of a process i, and more particularly its infinite *Input-Process-Output* loop. Since inputs are sampled at the beginning of such a cycle, and outputs are written at the end, the changes of the environment during the cycle have no effect whatsoever on the process phase of i. Thus, to avoid unnecessary interleaving between the environment and the different $_{\rm d}$SL processes, the part of the environment connected to this process should be *frozen* as long as process i is in its process phase. To achieve this, we integrate this part of the environment into the specification of process i (by means of *inlining*). Doing so allows the environment to change state only when the process reaches its input phase. The communications between the process and its part of the environment is done through shared variables (representing the I/O variables).

Modeling the processes The processes described in the semantics can be coded almost as-is into *Promela* processes using a straightforward syntactical transformation. Indeed, each $_{\rm d}$SL process can be translated into a *Promela* process that consists of an infinite loop performing the following steps : *input-process-output.* The input and output phases are interactions with the environment as described above. The process phase does the following steps :

- Input treatment : the triggering condition of all `WHEN`s owned by this process are considered consecutively, this to reacts to the input changes.
- Message treatment : reads the messages sent by other processes. This message treatment is translated into a loop reading a non-deterministic number of messages from the input channel for this process. Every time a message is read, `WHEN`s depending on the updated variable are checked for execution.

For this translation to work, we must have a special treatment for all assignments in the original $_{\rm d}$SL program. Indeed, an assignment `x:=e;` is translated into *Promela* as follows :

- `x:=e;`: the assignment is also performed in *Promela*.
- `ch1!x,e, ..., chk!x,e`: broadcasts the new value to all sites that use `~x`.
- consider all `WHEN`s conditioned on `x`.

According to the formal semantics, communications between the different processes are modeled using *Promela*'s `chan` and are kept reliable but not instantaneous.
In order to reduce the state space search, two changes with respect to the formal semantics are added in the *Promela* specification. First, the update messages caused by the assignment to a variable `x` are only sent to sites that use `x`, and not to all sites. The correctness of this optimization with respect to the formal semantics can easily be shown. Since a process not using `~x`, never accesses its local copy of `x`, the behavior of this process remains the same if we omit to send it the updates of `x`.
Second, we use *Promela*'s `atomic` construct to merge all transitions in the process phase of a particular process together into one meta transition[4]. To justify this reduction, observe that exterior processes can not have any effect on the behavior of the process phase and that its progression has no influence on other processes. Indeed, a process is only influenced by its FIFO channels and the environment, and neither of them are consulted during processing. On the other hand, the environment remains unchanged during this phase, and another process can only observe the modification of the FIFO channels when messages are sent. Although the introduction of `atomic` groups such messages together, this has no effect since the number of messages received in the input phase is non deterministic.
Caused by our asynchronous execution scheme which imposes that processes can only observe their own local variables, this reduction can be applied to any $_{\rm d}$SL program. Notice that the process phase contains most of the controller's behavior, and that this reduction is therefore substantial.

[4] Note that, since messages are sent during the process phase and FIFO channels might be full, the use of *Promela*'s `d_step` is prohibited

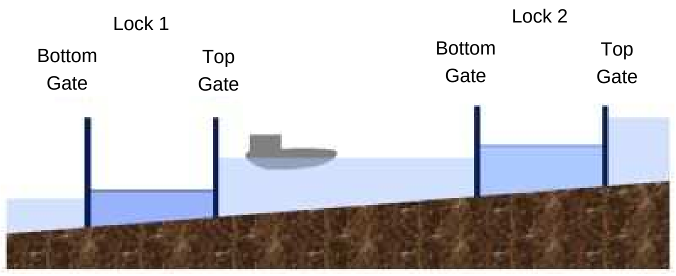

Fig. 3. Canal locks

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

Fig. 4. `WHEN` monitoring the command "open the bottom gate of lock2"

7. Case study : a canal lock controller

The problem To illustrate dSL concepts and the verification of a dSL program, we study the design of a controller for a system composed of two consecutive locks. As presented in figure 3, 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 or 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 all time. Whenever a command is introduced via the control panel, before taking the appropriate action, the controller must first check that it will not jeopardize the system, in which case, the action is not taken, and a red light on the control panel is switched on to indicate an error.

The solution The idea to implement the controller in dSL is the following. Whenever an order is given, a corresponding boolean variable `order_given` is set (there is an `order_given` variable for each gate and one for the water level of each lock). When receiving a command, the controller has to check that all the requirements are satisfied and, using those `order_given` variables, that no order on the checked gates and water levels are given (note that an order to close a gate can never violate a constraints). The `order_given` variables are, of course, reset when an order is completed. In this implementation, each command is monitored by a `WHEN` construct. As an example, figure 4 presents the `WHEN` monitoring the command "open the bottom gate of lock2" (the complete dSL source can be found in appendix B.1).

Note that for all the `order_given` variables, the '~' operator cannot be used. For example, in figure 4, if `lock2.top_gate.order_given` was tilded, when an order is given to open the bottom gate of lock2, the controller would check if `~lock2.top_gate.order_given` is false. However, in that case, because of communication delays, an order might have been given. The controller would then allow the bottom gate of lock2 to open while the top gate is ordered to open, which leads to a violation of the given constraints.

```
inline flip_flop_behavior(sensor_flipped, sensor_flopped, order_cmd, order_dir) {
   if :: order_cmd && sensor_flopped ->
         if :: order_dir == ORDER_TO_FLOP -> skip;
            :: order_dir == ORDER_TO_FLIP -> sensor_flopped = false;
         fi;
      :: order_cmd && !sensor_flopped && !sensor_flipped ->
         if :: order_dir == ORDER_TO_FLOP -> sensor_flopped = true;
            :: order_dir == ORDER_TO_FLIP -> sensor_flipped = true;
            :: skip;
         fi;
      :: order_cmd && sensor_flipped ->
         if :: order_dir == ORDER_TO_FLOP -> sensor_flipped = false;
            :: order_dir == ORDER_TO_FLIP -> skip;
         fi;
      :: skip;
   fi;
}
```

Fig. 5. Flip flop behavior

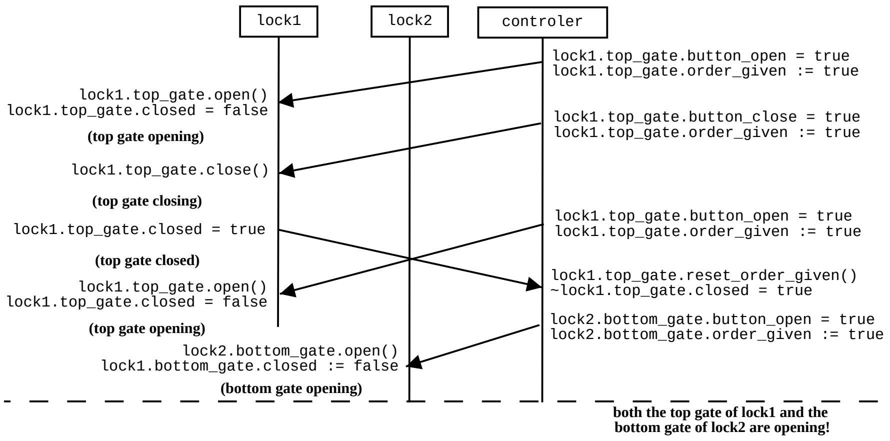
lock1
lock2
controler
lock1.top_gate.button_open = true
lock1.top_gate.order_given := true
lock1.top_gate.open()
lock1.top_gate.closed = false
(top gate opening)
lock1.top_gate.button_close = true
lock1.top_gate.order_given := true
lock1.top_gate.close()
(top gate closing)
lock1.top_gate.button_open = true
lock1.top_gate.order_given := true
lock1.top_gate.closed = true
(top gate closed)
lock1.top_gate.reset_order_given()
~lock1.top_gate.closed = true
lock1.top_gate.open()
lock1.top_gate.closed = false
(top gate opening)
lock2.bottom_gate.button_open = true
lock2.bottom_gate.order_given := true
lock2.bottom_gate.open()
lock1.bottom_gate.closed := false
(bottom gate opening)
both the top gate of lock1 and the bottom gate of lock2 are opening!

Fig. 6. Error trace

Translation into *Promela* We now explain how to translate the $_{\text{d}}$SL program for the canal lock controller into *Promela*. Remember that this translation depends on a given distribution. Here, we have considered several possible distributions, from the minimal distribution to the maximal distribution. For this particular application, the maximal distribution consists of 11 sites, where 4 of the 11 sites each monitor a single control button responsible for sending the command to close one of the gates. Results for this extreme configuration are hard to obtain since the number of processes makes the state space size explode. In the 7-sites distribution, the control panel, each gate and each water level is controlled by a single site, while the 3-sites distribution has the control panel on one site, and each lock controlled by another site. The 2-sites distribution has one site for both locks, and one for the control panel.

Note that each of these distributions with more sites is a refinement of a distribution with less sites which allows us to illustrate the simulation relation. Note however that in practice, due to theorem 1, the correctness of for example the 7-sites distribution is sufficient to prove the correctness of the 3-sites, 2-sites and 1-site distribution.

The gates and the water levels are modeled using a *flip-flop* behavior (see figure 5) that has four states : flipped (1,0), flopped(0,1), flipping(0,0) and flopping(0,0). It can receive an order to flip, to flop or to do nothing. The behavior is obvious, and can easily be adapted for the gates (flipped ≡ opened, flopped ≡ closed) as for the water level (flipped ≡ up, flopped ≡ down). A nondeterministic choice makes the gate (and the water) move from opened (up) to closed (down) by allowing the model to stay in the flipping, respectively flopping state. Modeling the operator is straightforward, a nondeterministic choice lets the operator choose one of the twelve buttons (`lock{1/2}.{top/btm}_gate.btn_{open/close}` and `lock{1/2}.btn_{empty/fill}`).

```
WHEN lock2.bottom_gate.btn_open and not disabled THEN
  disabled := true;
  launch open_bottom_gate_lock2;
END_WHEN

SEQUENCE open_bottom_gate_lock2()
VAR
  check : bool;
END_VAR
  check := (lock2.top_gate.closed AND lock2.water_down);
  check := (check AND lock1.top_gate.closed);
  IF check THEN
    not_allowed_led := false;
    LAUNCH lock2.bottom_gate<-open();
  ELSE
    not_allowed_led := true;
  END_IF;
  disabled := false;
END_SEQUENCE
```

Fig. 7. `WHEN` / `SEQUENCE` monitoring the command "open the bottom gate of lock2"

Problem in the locks controller! At first glance, this implementation seems to work. However, after modeling it in *Promela* as explained in section 6 and using *Spin* model checker to verify the given constraints, we found out that it is faulty. Indeed, as shown in figure 6, two consecutive gates (top gate of lock 1 and bottom gate of lock 2) can be opened at the same time. In this case, three orders are given to the top gate of lock 1: an order to open, followed by an order to close (before the gate is completely opened) and finally an order to open. Because of communication, the `reset_order_given()` and the value of `~lock1.top_gate_closed` are delayed (respectively because of the `launch` and '~'). Therefore when the order to open the bottom gate of lock 2 is given, the controller believes that the top gate of lock 1 is closed and that no order has been given to it. This allows the opening of the bottom gate of lock 2, which violates the constraints.

An easy way to correct this, would be to allow a command to a gate (or a water level) only if its `order_given` is false (in other words, only allowing one order at a time). However, this would not be a viable solution. Indeed, imagine a boat breaks down while the gate is closing, the controller would not allow to open a gate until it is completely closed, and the boat would be crushed down! So, instead of blocking all commands while an ordered is processed, we disable the commands only during the time needed to verify the (distributed) constraints. To achieve this, a sequential execution checks that the issued command can be executed, by migrating the condition to all intervening sites. As illustrated in figure 7, this is done by means of a `SEQUENCE` construct that evaluates, in the local variable `check`, that all the conditions are satisfied. In the example of figure 7, the first part of the constraint (`check := (lock2.top_gate.closed and lock2.water_down);`) will be evaluated on the site where `lock2` is localized, then the value of `check` will be migrated to the site where `lock1` is localized to evaluate the second part (`check := (check and lock1.top_gate.closed);`). Since the control panel is disabled during this task, we can be sure that the variable `check` is true if and only if the constraints are satisfied, in which case, the corresponding action(s) is (are) taken. This introduces the need for classical distributed systems mechanisms such as semaphores.

Verification Two properties were checked on the model. the first property expresses all the constraints presented in section 7. It is expressed in LTL using the formula `[] !global_bad` where `global_bad` is defined in *Spin* as follows:

```
#define global_bad ( (!lock1_top_gate_closed && !lock1_bottom_gate_closed) ||
                     (!lock1_top_gate_closed && !lock2_bottom_gate_closed) ||
                     (!lock2_bottom_gate_closed && !lock2_top_gate_closed) ||
                     (!lock1_bottom_gate_closed && !lock1_water_down)      ||
                     (!lock1_top_gate_closed && !lock1_water_up)           ||
                     (!lock2_bottom_gate_closed && !lock2_water_down)      ||
                     (!lock2_top_gate_closed && !lock2_water_up)           ||
                   )
```

Sites	Btn	Channels	Property	P.O.	Verified	Time	States	Memory
1	2	Instant	`global_bad`	yes	$\surd$	0:03.25	1.41e+04	1.432 MB
2	2	Instant	`global_bad`	yes	$\times$	0:14.52	8.37e+04	3.064 MB
3	2	Instant	`global_bad`	yes	$\times$	0:08.00	9.54e+04	3.569 MB
7	2	Instant	`global_bad`	yes	$\times$	0:09.74	7.82e+04	3.645 MB
11	2	Instant	`global_bad`	yes	$\times$	5:38.48	1.41e+06	68.557 MB
1	1	Normal	`middle_bad`	yes	$\surd$	0:07.80	3.98e+04	2.457 MB
2	1	Normal	`middle_bad`	yes	$\surd$	0:43.22	1.29e+05	4.908 MB
3	1	Normal	`middle_bad`	yes	$\surd$	0:37.13	1.39e+05	5.311 MB
7	1	Normal	`middle_bad`	yes	$\times$	21:14.36	1.91e+06	77.581 MB
11	1	Normal	`middle_bad`	yes	$\times$	0:54.93	2.07e+05	10.383 MB
1	1	Instant	`middle_bad`	no	$\surd$	0:00.15	437	0.409 MB
1	1	Instant	`middle_bad`	yes	$\surd$	0:00.15	437	0.409 MB
2	1	Instant	`middle_bad`	no	$\surd$	0:01.04	8133	0.709 MB
2	1	Instant	`middle_bad`	yes	$\surd$	0:01.08	8132	0.709 MB
3	1	Instant	`middle_bad`	no	$\surd$	0:00.72	10136	0.805 MB
3	1	Instant	`middle_bad`	yes	$\surd$	0:00.73	10133	0.805 MB
7	1	Instant	`middle_bad`	no	$\surd$	0:11.46	146401	6.206 MB
7	1	Instant	`middle_bad`	yes	$\surd$	0:11.96	146338	6.206 MB
11	1	Instant	`middle_bad`	no	N/A	TIMEOUT	2.10+07	1 GB
11	1	Instant	`middle_bad`	yes	$\surd$	18:03.57	5.43e+06	261.717 MB

Fig. 8. Results of the verification of `global_bad` and `middle_bad` properties on the first (faulty) version of the canal lock controller.

The second property expresses only the fact that the two middle gates (i.e top gate of lock1 and bottom gate of lock 2) of the locks are opened at the same time. It is expressed in LTL using the formula `[] !middle_bad` where `middle_bad` is defined in *Spin* as follows:

```
#define middle_bad (!lock1_top_gate_closed && !lock2_bottom_gate_closed)
```

The verification was made using *Spin* version 4.2.4, on a 3 GHz Intel Xeon machine with 2GB of memory. Some representative results corresponding to the verification of these properties on the first (faulty) version of the canal lock controller are shown in figure 8. The first column indicates the number of execution sites, respectively 1 (minimal), 2, 3, 7 and finally 11 (maximal). The second column indicates the maximum number of times each button can be pressed. The next column indicates whether or not messages are taken from their queues as soon as possible. The fourth and fifth columns shows respectively the property that was checked, and whether or not partial order reduction was used. The four remaining columns show respectively, whether the property was verified or not, the time needed for the verification, the number of states explored and the memory usage. First of all, by examining the first two sets of experiments, we can observe that the results are coherent with theorem 1. That is, if the error in figure 6 is found in a distribution (indicated by $\times$ in column 6), it is also found in more refined distributions. Finally, in the third set of experiments, we can observe that the gain of partial order reduction on time and memory used is limited. The gain is substantial only for the maximal distribution. This is because, in the specification, we intensively use `atomic` blocks to reduce the state space, therefore reducing the efficiency of the partial order reduction.

8. Conclusions

In this paper, we presented $_d$SL, an environment to design industrial distributed process control systems. We pointed out the main advantages of using $_d$SL, compared to other approaches like synchronous languages, design languages (like B or Unity) or solutions using shared variables. Among the advantages, $_d$SL is first a programming language which extends the ST standard, guaranteeing its use in practice. $_d$SL also provides automatic code distribution and includes two features (`SEQUENCE` and tilded variables) which allows the programmer to be isolated from the physical distribution of the system. Moreover, we defined simple operational semantics, parametrized by the distribution used for its execution. A $_d$SL program is seen as a composition of asynchronous processes communicating through FIFO channels. The main result states that

a more refined distribution can simulate a less refined one, which in its turn allows to define a lattice of semantics, where every next-free LTL property is preserved from its maximal element (most refined) to its minimal element (single site).
We also showed how this semantics can easily be verified using the explicit state model checker *Spin*, using a translation into its input language *Promela*. We showed that *"real"* size programs (here, a design with more than 60 boolean variables) can be model checked using a fair amount of time and memory. This is due to the special structure of dSL programs, where the processing phase of the input-process-output cycle can safely be handled in a single transition in the state space.
In the future, we would like to automate the entire translation of dSL to *Promela*, and to apply various algorithms such as slicing and abstraction to reduce the complexity of the resulting model. In an effort to simplify the designer's task, we must find an intuitive and accessible way to model the environment. Since control of industrial processes often uses standard devices, we could provide the designer with a library of common pre-modeled and parametrized environment behaviors. We should also provide a dSL library of classical distributed mechanisms (i.e. semaphores, mutex).

Acknowledgments The authors gratefully acknowledge the many helpful suggestions of the reviewers which greatly improved the overall quality of the paper.

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* ISBN 0-521-49619-5. Cambridge University Press, UK, 1996.

[Aub97] P. Aubry. *Mises en oeuvre distribues 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.

[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.

[BMS97] F. Bonfattti, P.D. Monari, and U. Sampieri. *IEC 1131-3 programming methodology. Software engineering methods for industrial automated systems.* ISBN 2-9511585-0-5. CJ International Editions, 1997.

[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.

[DeWGM05] Bram De Wachter, Alexandre Genon, and Thierry Massart. From static code distribution to more shrinkage for the multiterminal cut. Technical Report 537, ULB, 2005. To appear in proceedings of WEA05.

[DeWMM03] Bram De Wachter, Cédric Meuter, and Thierry Massart. dsl: An environment with automatic code distribution for industrial control systems. In M. Papatriantafilou and Ph. Hunel, editors, *"Principles of Distributed Systems: 7th International Conference*, volume 3114 of *LNCS*, pages 132–145, La Martinique, December 2003. Springer-Verlag.

[DJP$^+$94] Elias Dahlhaus, David S. Johnson, Christos H. Papadimitriou, P. D. Seymour, and Miha lis Yannakakis. The complexity of multiterminal cuts. *SIAM J. Comput.*, 23(4):864–894, 1994.

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

[Gen04] Alexandre Genon. On the verification of dsl, a language to design distributed industrial control systems. Technical report, U.L.B., September 2004.

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

[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.

[Hol03] Gerard J. Holzmann. *The Spin Model Checker - Primer and Reference Manual.* Addison-Wesley Publ., 2003.

[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.

[KMC88] J. Misra K. Mani Chandy. *Parallel program design: a foundation.* Addison-Wesley Longman Publishing Co., Inc., 1988.

[LGLL91] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real time applications with signal. Proceedings of the IEEE, 79(9):1321-1336, September 1991.

[LMW04] S. Leue, R. Mayr, and W. Wei. A scalable incomplete test for the boundedness of uml rt models. In K. Jensen and A. Podelski, editors, *Proceedings: 10th International Conference, TACAS 2004*, volume 2988 of *LNCS*, pages 327–341, Barcelona, Spain, March 2004. ETAPS 2004, Springer-Verlag.

[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.

[SEM03] Alin Stefanescu, Javier Esparza, and Anca Muscholl. Synthesis of distributed algorithms using asynchronous automata. In R. Amadio and D. Lugiez, editors, *Proceedings of the 14th International Conference on Concurrency Theory (CONCUR'03)*, volume 2761, pages 27–41. Springer, September 2003.

A. Proof of Theorem 1

Definition 11 (Workload distribution). Let $\omega, \omega_1, \omega_2$ be three workloads (i.e. sequences of instructions, including abstract instructions). We have that (ω_1, ω_2) is a distribution of ω, noted $(\omega_1, \omega_2) \succ \omega$, if and only if

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

♦

Lemma 1. Given a well-formed dSL program P and $D = (V_1, ..., V_n)$, $D' = (V_1', ..., V_n', V_{n+1}')$ 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 $(E_D^P)_n = (W_n)$, $(E_{D'}^P)_n = (W_n')$ and $(E_{D'}^P)_{n+1} = (W_{n+1}')$, we have :

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

■

Lemma 2 (One-split simulation). Given a well-formed dSL 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 that $V_n = V_n' \cup V_{n+1}'$. We have that $[\![P]\!]_{D'}$ simulates $[\![P]\!]_D$:

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

Proof. We define a relation $\mathcal{R} \subseteq \mathcal{G}_D^P \times \mathcal{G}_{D'}^P$ such that if $G = ((\omega_1^G, \nu_1^G, \phi_1^G), (\omega_2^G, \nu_2^G, \phi_2^G), ..., (\omega_n^G, \nu_n^G, \phi_n^G))$ and $G' = ((\omega_1^{G'}, \nu_1^{G'}, \phi_1^{G'}), (\omega_2^{G'}, \nu_2^{G'}, \phi_2^{G'}), ..., (\omega_n^{G'}, \nu_n^{G'}, \phi_n^{G'}), (\omega_{n+1}^{G'}, \nu_{n+1}^{G'}, \phi_{n+1}^{G'}))$, $(G, G') \in \mathcal{R}$ if and only if:

1. $(\omega_i^G, \nu_i^G, \phi_i^G) = (\omega_i^{G'}, \nu_i^{G'}, \phi_i^{G'})$, $\forall i \in [1..n-1]$
2. $\phi_n^{G'} = \phi_{n+1}^{G'} = \phi_n^G$
3. $\forall x \in (V_n' \cup \widetilde{Var(P)} \cup OldCond(W_n'))$, $\nu_n^G(x) = \nu_n^{G'}(x)$
4. $\forall x \in (V_{n+1}' \cup \widetilde{Var(P)} \cup OldCond(W_{n+1}'))$, $\nu_n^G(x) = \nu_{n+1}^{G'}(x)$
5. $(\omega_n^{G'}, \omega_{n+1}^{G'}) \succ \omega_n^G$

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

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

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

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

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

The transition $G \xrightarrow{a} H$ can be derived from one of 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 $(G)_i = (G')_i$. Since the broadcast can be fired by $(G)_i$ in $[\![P]\!]_D$, it can also be fired by $(G')_i$ in $[\![P]\!]_{D'}$. It follows directly that:

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

The message is added to all FIFO channels. Therefore, FIFO channels in H and H' are identical. 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 $\texttt{BCAST(x)}$ 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' are identical. It is then easy to see that $(H, H') \in \mathcal{R}$.

   (b) $\omega_n^{G'} = \omega_n^{H'}$ and $\omega_{n+1}^{G'} = \texttt{BCAST(x)};\omega_{n+1}^{H'}$, the 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 $(G')_i = (G)_i$. Since the local transition can be fired by $(G)_i$ in $[\![P]\!]_D$, it can also be fired by $(G')_i$ in $[\![P]\!]_{D'}$. It follows that:

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

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

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

   [Cycle Start] In this case, we have $\omega_n^G = \epsilon$, $(a = \tau)$. 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 us 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 :

$$\phi_n^H = \psi_1 \cdot \diamond \cdot \psi_2 (\text{where } \phi_n^G = \psi_1 \cdot \psi_2) \wedge \nu_n^G = \nu_n^H$$
$$\phi_n^{H'} = \psi_1 \cdot \diamond \cdot \psi_2 (\text{where } \phi_n^{G'} = \psi_1 \cdot \psi_2) \wedge \nu_n^{G'} = \nu_n^{H'}$$
$$\phi_{n+1}^{H'} = \psi_1 \cdot \diamond \cdot \psi_2 (\text{where } \phi_{n+1}^{G'} = \psi_1 \cdot \psi_2) \wedge \nu_{n+1}^{G'} = \nu_{n+1}^{H'}$$

   Indeed, the valuations on the variables remain the same when the cycle start rule is applied. For the FIFO channels, the end of message symbol $(\diamond)$ is inserted at the same place in all cases. Let us look at ω_n^H, by lemma 1, we know that every input/output/`WHEN` treatment instruction inserted in ω_n^H will be inserted either in $\omega_n^{H'}$ or $\omega_{n+1}^{H'}$. More formally, this gives :

$$(Sample(Var^{in}(P) \cap V_n'), Sample(Var^{in}(P) \cap V_{n+1}')) \succ Sample(Var^{in}(P) \cap V_n)$$

$$(Write(Var^{out}(P) \cap V_n'), Write(Var^{out}(P) \cap V_{n+1}')) \succ Write(Var^{out}(P) \cap V_n)$$

$$(Treat(W_n'), Treat(W_{n+1}')) \succ Treat(W_n)$$

Thus, we have $(\omega_n^{H'}, \omega_{n+1}^{H'}) \succ \omega_n^H$ and $G' \xrightarrow{\tau \cdot \tau} H'$ and $(H, H') \in \mathcal{R}$.

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

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

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

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

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

[Message treatment] In this case, we have $\omega_n^G = MSG; \eta_n^H$, $(a = \tau)$. 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 same rule to $(G')_n$ and $(G')_{n+1}$, we then have a sequence of transitions :

$$\begin{array}{l} G' \xrightarrow{\tau} G'' \xrightarrow{\tau} H' \\ \text{Where} \\ (G')_n \xrightarrow{\tau} (G'')_n \wedge \forall i \neq n (G')_i = (G'')_i \\ (G'')_{n+1} \xrightarrow{\tau} (H')_{n+1} \wedge \forall i \neq n+1 (G'')_i = (H')_i \end{array}$$

If $G \xrightarrow{\tau} H$ results from the application of the end of message treatment rule, then we have that $\phi_n^G = \diamond \cdot \psi$. By definition of $\mathcal{R}$, we have that $\phi_n^G = \phi_n^{G'} = \phi_{n+1}^{G'}$. Thus, we may simply apply the end of message treatment rule to $(G')_n$ and $(G')_{n+1}$.

If the message treatment rule is applied, then we have $\phi_n^G = (x, v) \cdot \phi'$, as $\phi_n^{G'} = \phi_{n+1}^{G'} = \phi_n^G$, the same message can be read by the three processes, the valuation will be modified accordingly and the list of whens will be inserted in the workload. We have :

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

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

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

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

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

(a) if $\omega_n^{G'} = \texttt{x} := \texttt{e}; \eta_n^{H'}$ 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 us 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 constraints induced by the `WHEN`s are violated. Thus, D' would not be a distribution and we must have $W'_{n/x} = W_{n/x}$. In conclusion, we have :

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

As $(\eta_n^{H'}, \omega_{n+1}^{H'}) \succ \eta_n^{H'}$, 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 the remaining cases ($x \in V(p)$ or $x \in OldCond(P)$), we have $G' \xrightarrow{\tau} H'$ and, therefore, $(H, H') \in \mathcal{R}$.

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

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

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

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

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

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

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

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

Finally, it easy to prove that $(G_D^0, G_{D'}^0) \in \mathcal{R}$. Indeed, according to definition 7, both in G_D^0 and $G_{D'}^0$, the workloads are empty, the valuations are assigning all variables to $\bot$ and the Fifo channel are empty. Therefore, by definition of $\mathcal{R}$ and by definition 11, we have $(G_D^0, G_{D'}^0) \in \mathcal{R}$ and :

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

■

Lemma 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 $D_1, ..., D_n$ (with $n = l - k + 1$), such that

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

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

Theorem 1 (Simulation). Given a well-formed dSL 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 2, lemma 3, and the transitivity of the simulation relation. ■

B. Lock controller : dSL source

B.1. First attempt

```
CLASS Gate
  motor_direction, motor_command, opened, closed, order_given : BOOL;
  button_open, button_close : BOOL;
END_CLASS

CLASS Lock
  water_up, water_down, water_command, water_direction, water_order_given : BOOL;
  bottom_gate, top_gate : GATE;
  button_fill, button_empty : BOOL;
END_CLASS

GLOBAL_VAR
  lock1, lock2                          : Lock;
  not_allowed_led                       : BOOL;
END_VAR

(* Gates *)
METHOD GATE::move(direction : BOOL)
  self.motor_direction := direction;
  self.motor_command := TRUE;
END_METHOD

METHOD GATE::reset_order_given()
  self.order_given := FALSE;
  not_allowed_led := FALSE;
END_METHOD

(* Equivalent to WHEN G.closed OR G.opened THEN ... for every object G of class GATE *)
WHEN IN GATE self.closed OR self.opened THEN       (* W1 = self -> lock1.bottom_gate *)
  self.motor_command := FALSE;                     (* W2 = self -> lock1.top_gate    *)
  LAUNCH self<-reset_order_given();                (* W3 = self -> lock2.bottom_gate *)
END_WHEN                                           (* W4 = self -> lock2.top_gate    *)

(* Locks *)
METHOD LOCK::water_move(direction : BOOL)
  IF NOT self.water_down THEN
    self.water_command := TRUE;
    self.water_direction := direction;
  END_IF;
END_METHOD

METHOD LOCK::reset_water_order_given()
  self.water_order_given := FALSE;
  not_allowed_led := FALSE;
END_METHOD

WHEN IN LOCK self.water_up OR self.water_down THEN   (* W5 = self -> lock1 *)
  self.water_command := FALSE;                       (* W6 = self -> lock2 *)
  LAUNCH self<-reset_water_order_given();
END_WHEN
```

```
WHEN lock1.bottom_gate.button_open THEN              (* W7 *)
  IF (~lock1.top_gate.closed) AND (NOT lock1.top_gate.order_given) AND
     (~lock1.water_down) AND (NOT lick1.water_order_given)
  THEN
    not_allowed_led := FALSE;
    lock1.bottom_gate.order_given := TRUE;
    LAUNCH lock1.bottom_gate<-move(TRUE);  (*open*)
  ELSE
    not_allowed_led := TRUE;
  END_IF;
END_WHEN

WHEN lock1.top_gate.button_open THEN                 (* W8 *)
  IF (~lock1.bottom_gate.closed) AND (NOT lock1.bottom_gate.order_given) AND
     (~lock2.bottom_gate.closed) AND (NOT lock2.bottom_gate.order_given) AND
     (~lock1.water_up) AND (NOT lock1.water_order_given)
  THEN
    not_allowed_led := FALSE;
    lock1.top_gate.order_given := TRUE;
    LAUNCH lock1.top_gate<-move(TRUE);   (*open*)
  ELSE
    not_allowed_led := TRUE;
  END_IF;
END_WHEN

WHEN lock1.bottom_gate.button_close THEN             (* W9 *)
  LAUNCH lock1.bottom_gate<-move(FALSE);  (*close*)
END_WHEN

WHEN lock1.top_gate.button_close THEN                (* W10 *)
  LAUNCH lock1.top_gate<-move(FALSE);     (*close*)
END_WHEN

WHEN lock1.button_fill THEN                          (* W11 *)
  IF (~lock1.bottom_gate.closed) AND (NOT lock1.bottom_gate.order_given) AND
     (~lock1.top_gate.closed) AND (NOT lock1.top_gate.order_given)
  THEN
    not_allowed_led := FALSE;
    lock1.water_order_given := TRUE;
    LAUNCH lock1<-water_move(TRUE);
  ELSE
    not_allowed_led := TRUE;
  END_IF;
END_WHEN

WHEN lock1.button_empty THEN                         (* W12 *)
  IF (~lock1.bottom_gate.closed) AND (NOT lock1.bottom_gate.order_given) AND
     (~lock1.top_gate.closed) AND (NOT lock1.top_gate.order_given)
  THEN
    not_allowed_led := FALSE;
    lock1.water_order_given := TRUE;
    LAUNCH lock1<-water_move(FALSE);
  ELSE
    not_allowed_led := TRUE;
  END_IF;
END_WHEN

WHEN lock2.bottom_gate.button_open THEN              (* W13 *)
  ... (* Same as W8, replace lock1 with lock2 ; switch top, bottom; switch up, down *)

WHEN lock2.top_gate.button_open THEN                 (* W14 *)
  ... (* Same as W7, replace lock1 with lock2 ; switch top, bottom; switch up, down *)

WHEN lock2.bottom_gate.button_close THEN             (* W15 *)
  LAUNCH lock2.bottom_gate<-move(FALSE);  (*close*)
END_WHEN
```

```
WHEN lock2.top_gate.button_close THEN                (* W16 *)
  LAUNCH lock2.top_gate<-move(FALSE);     (*close*)
END_WHEN

WHEN lock2.button_fill THEN                          (* W17 *)
  ... (* Same as W11, replace lock1 with lock2 *)

WHEN lock2.button_empty THEN                         (* W18 *)
  ... (* Same as W12, replace lock1 with lock2 *)

(* main program *)
SEQUENCE init
  not_allowed_led := FALSE;
END_SEQUENCE
```

B.2. Second attempt

```
(* Same as attempt 1, without the order given instructions : *)

CLASS GATE ... END_CLASS
CLASS LOCK ... END_CLASS

METHOD GATE::open() ... END_METHOD
METHOD GATE::close()... END_METHOD
METHOD GATE::reset_order_given() ... END_METHOD
WHEN IN GATE self.closed OR self.opened THEN ... END_WHEN

METHOD LOCK::empty() ... END_METHOD
METHOD LOCK::fill() ... END_METHOD

(* Commands on lock1 *)
WHEN lock1.bottom_gate.button_open AND NOT disabled THEN
  disabled := true;
  LAUNCH open_bottom_gate_lock1;
END_WHEN

SEQUENCE open_bottom_gate_lock1
VAR
  check : bool;
END_VAR
  check := (lock1.top_gate.closed AND lock1.water_down);
  IF check THEN
    not_allowed_led := FALSE;
    LAUNCH lock1.bottom_gate<-open();
  ELSE
    not_allowed_led := TRUE;
  END_IF;
  disabled := false;
END_SEQUENCE

WHEN lock1.top_gate.button_open AND NOT disabled THEN
  disabled := true;
  LAUNCH open_top_gate_lock1;
END_WHEN

SEQUENCE open_top_gate_lock1
VAR
  check : bool;
END_VAR
  check := (lock1.bottom_gate.closed AND lock1.water_up);
  check := (check AND lock2.bottom_gate.closed);
  IF check THEN
    not_allowed_led := FALSE;
    LAUNCH lock1.top_gate<-open();
  ELSE
    not_allowed_led := TRUE;
  END_IF;
  disabled := false;
END_SEQUENCE

WHEN lock1.bottom_gate.button_close THEN
  LAUNCH lock1.bottom_gate<-close();
END_WHEN

WHEN lock1.top_gate.button_close THEN
  LAUNCH lock1.top_gate<-close();
END_WHEN

WHEN lock1.button_fill AND NOT disabled THEN
  disabled := true;
  LAUNCH fill_lock1;
END_WHEN

SEQUENCE fill_lock1
VAR
  check : bool;
END_VAR
  check := (lock1.top_gate.closed AND lock1.bottom_gate.closed);
  IF check THEN
    not_allowed_led := FALSE;
    LAUNCH lock1<-fill();
  ELSE
    not_allowed_led := TRUE;
  END_IF;
  disabled := false;
END_SEQUENCE

WHEN lock1.button_empty AND NOT disabled THEN
  disabled := true;
  LAUNCH empty_lock1;
END_WHEN

SEQUENCE empty_lock1
  (* same as fill_lock1, replace empty() with fill() *)
...

(* Commands on lock2 : same as commands on lock1,
   replace lock1 with lock2,
   switch top and bottom,
   switch up and down. *)
...

SEQUENCE init
  not_allowed_led := FALSE;
SEQUENCE
```