dSL verification with *Spin*, A case study

Bram De Wachter Thierry Massart
Cédric Meuter

Brussels Free University, Computer Science Department

August 18, 2003

Abstract

In this paper, we first give an overview of the concepts and environment of dSL, a simple imperative and event driven language designed to program distributed industrial control systems. The advantage of dSL is to provide a transparent code distribution using low level mechanisms. The behavior of the synthesized distributed system can therefore be formally modeled or easily monitored. We show that another advantage is the possibility to be able to verify systems designed with dSL. As an example, we show how dSL can be used to design the control system of two canal locks and we use the *Spin* tool to prove correctness of the system.

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,...) which handle the communication aspects and allows the programmer to concentrate on the functionality aspects of the system. Unfortunately, these solutions are generally quite heavy, and completely hide all the communications aspects, making the monitoring of such systems difficult.

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

We show that another advantage is the possibility to be able to verify systems designed with dSL. As an example, we show how dSL was used to design the control system of two canal locks. We show how a translation can be done between dSL and *Promela*, the language used by the *Spin* tool, and present the verifications we have done with *Spin* to prove correctness of our system.

The paper is organized as follows. Section 2 describes dSL concepts together with a simplified syntax and an informal semantics that capture the most of dSL's possibilities. In section 3, we discuss dSL semantics. In Section 4 we present our simple control system of locks. Section 5 presents the translation in *Promela* of our dSL program. In section 6, we present the results obtained with *Spin*. Finally, some concluding remarks are given in section 7.

2 The dSL concept

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

dSL [1] is both a programming language used to design industrial control systems and a programming environment, mainly a compiler which synthesizes the actual control systems.

dSL is a simple imperative language with static variables; a variable can be (1) internal to the program (2) linked to an input (sensor) or (3) linked to an output (actuator). dSL is event driven, i.e., an event is specified by the change in some variable value. For instance, `when x >= 0 then run_motor1();` will trigger `run_motor1()` every time the variable x switches from a negative to a positive value.

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

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

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

dSL example

To illustrate the dSL concepts, let us give a simplified example of a program that receives input values through an input variable `temp`, linked for instance to a temperature sensor and switches on (off) a `heater` when the temperature is below 0 ^oC (above 20 ^oC). Two indicators in a control panel (`led` and `alarm`) are also updated following the state of the heater and the number of times the heater is turned from one state to the other, respectively. The program will be used in a physical configuration counting 2 sites; but this is not directly mentioned in the program. The dSL program of figure 1 corresponds to this simple control system.

A dSL program contains 5 elements. (1) global variables declarations including all I/O variables (2) method definitions (3) when instructions (4) sequence definitions and (5) an initialization. A simplified grammar of the dSL syntax can be found in appendix A.

Note that dSL has limited Object-Oriented features. Variables and methods are therefore structured into object definitions. We will not detail this aspect further in this paper.

Example of figure 1 does not contain any method.

The `when` construct allows to formulate an event-handler; it is made up of two parts : 1. (*expr*) The condition that must change from `false` to `true` enabling the handler and 2. (*instruction_ list*) the code that has to be executed in that case.

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

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

```
global_var
  led, heater : output_byte;
  alarm       : output_byte;
  temp        : input_byte;
  maintenance : bool;
  control     : int;
end_var

sequence set_heater(state)
(* Make the led's state correspond to
   the heater's action *)
   control := control+1;
   if control == 1000 then
      control := 0;
      maintenance := true;
   end_if
   led := state;
   heater := state;
end_sequence

when maintenance then
  alarm := true;
(* ... *)
end_when

when temp < 0 then
(* Turn on the heater *)
   launch set_heater (1);
end_when

when temp > 20 then
(* Turn off the heater *)
   launch set_heater (0);
end_when

program
   control := 0;
(* Initially turn off heater and led indicator *)
   launch set_heater (0);
end_program
```

Figure 1: simple dSL program

atomic and sequential code

The design of dSL has been dictated by the execution paradigm used in the world of industrial process control that requires a immediate reaction to events and their instantaneous treatment. In practice, by default, this forbids any hidden 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 dSL distinguishes between

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

By default, the code attached to a *when* is *event-driven* and therefore must be local to a given site. To relax this constraints, two mechanisms have been defined: (1) the *launch* instruction allows to start *sequential* code[3], and (2) portions of *event-driven* code may indirectly know the

[3] A method (which is atomic) can also be launched. This results in a delayed execution of an atomic code

Site 1	Site 2

label0:
control:=control+1;
if control == 1000 then
control := 0;
start 2, *label1*
stop
label2:
end_if
send 2, *state*
start 2, *label3*
stop
label4:
heater:=state;
stop
	label1:
	maintenance:=1;
	start 1, *label2*
	stop
	label3:
	receive *state* from **1**
	led:=state;
	start 1, *label4*
	stop

Figure 2: Distributed code of the procedure set_heater

value of variables using *tilded* variables (also called asynchronous variables). For instance, the instruction `y := ~x` assigns to the local variable `y` the last value received in the local site of the distant variable `x`; even if the value of `x` has changed since then. More precisely, the instruction `y := ~x` will stand for `y :=` $\tilde{x}_i$ where y is local to the site S_i. The variable $\tilde{x}_i$ is used in an event-driven code when the corresponding variable `x` cannot be local to this code (information given by the compiler when it tries to distribute the code. *Tilded* variables must be used with care : only when the exact value of a variable which cannot be local is not needed (e.g. temperature which evolves slowly) or if the program is built such that it is known that the *tilded* value is equal to the real one.

In dSL*sequential* code is defined through the *sequence* instruction. A *sequence* cannot have more than one instance executed simultaneously.

distributed code

In our example, we suppose the I/O variables `led, alarm` are governed by the first controller (site [1]), while `heater` and `maintenance` are localized on the second controller (site [2]); these information are given by the developer in a *localization table*.

Note that if I/O variables and instructions using them are connected to a given site, the localization for other variables and instructions is left to the compiler that should be decided before generating the distributed code. This distributed code should be equivalent to the initial dSL program in the sense that the possible interactions between the system and the environment should respect, all along its execution, what is specified by the initial dSL program. For instance, if the compiler connects the variable `control` and all "free" instructions to the site 1, then the resulted distributed code for the `set_heater` method would be as follows.

```
[1]       control := control+1;
[1]       if control == 1000 then
[1]          control := 0;
[2]          maintenance := 1;
[1]       end_if
[2]       led := state;
[1]       heater := state;
```

where the value between [and] identifies the localization of each instruction. Remark that these are indications used to point out the distribution of this particular dSL program, and are never part of any real dSL program (i.e. they are not present in the dSL grammar). The figure 2 shows the corresponding distributed code. We can also notice there that the value of `state` is transmitted from site 1 to site 2 where it is used to update the variable `led`.

3 Semantics

The semantics of $_d$SL is introduced more formally in this section allowing us to motivate the translation of $_d$SL programs to *Promela*. This section explains in more detail the insides of $_d$SL's *sequence*, *launch* and *tilded* variable. We will therefore skip a complete and formal review of the language by describing well known program issues like method call, control flow and expression evaluation.

As we have already seen, the main difference between $_d$SL and any common imperative programming language results from its distributed characteristics. To describe the semantics of $_d$SL independently from the actual distribution (materialized as a *localization table* described in section 2), we introduce the notion of *maximal distributability*, which expresses the most liberal configuration on which a given $_d$SL program could ever run. Indeed, due to *atomic* code, some instructions and hence variables must be sticked together in a single site; but in the other case, the code may be distributed. Using the $_d$SL semantics, we can show that there is a maximal possible distribution. The principle of maximal distributability expresses that maximal possible distribution allows the most possible behaviors. Verifying safety properties on this configuration will induce safety for any other distribution. In this section, we will show how this maximal distribution is obtained.

We will define a partition $(P_1, \cdots, P_k)$ of the event-driven instructions of a $_d$SL program, where the instructions in each element of the partition govern a set of variables. This partition may be used to describe k independent processes or *sites*. Since the partition is maximal, a given $_d$SL program may therefore contain less but not more sites than allowed by the partition.

Finding Independent Processes in $_d$SL

Definition 1 *Let $Instr(W_i)$ denote all instructions that may be reached in the body and condition of a given when W_i, including all code reachable from that when. This includes all instructions in the body of the when, together with all whens that may be triggered trough assignment and code reached trough synchronous method call.*

Definition 2 *Let $Var(instr)$ be the non tilded global variables used (read) and defined (written) in instr, and $Var(I) = \cup_{i \in I} Var(i)$, $Var(W_i) = Var(Instr(W_i))$.*

We can now express the locality of instantaneous code with the condition :

$$\forall W_i, W_{j \neq i} :$$

$$Var(W_i) \cap Var(W_j) \neq \phi \Rightarrow \exists l : Instr(W_i) \cup Instr(W_j) \subseteq P_l$$

To obtain a correct partition, we augment with the condition that

$$\forall W_i : \exists l : Instr(W_i) \subseteq P_l; \forall Instr : \exists l : Var(instr) \subseteq Var(P_l)$$

Now, retain the maximal partition of the event-driven instructions, $P_1, \cdots, P_k$, from all partitions that satisfy these conditions and such that

$$\cup_i Instr(W_i) = \cup_j P_j$$

Each P_i defines an independent process that will execute the instructions in P_i, and governs the variables in $Var(P_i)$.

To allow a given process to have a view outside its boundaries imposed by the locality constraints, asynchronous variables provide a *nearly* correct value for a variable that is governed by another process. Intuitively, if a process S_1 governs x, and another process S_2 needs a nearly correct value for x, it uses a tilded version $\tilde{x}$ that keeps S_1 and S_2 as independent processes. Each time S_1 changes x, S_2 will be notified of the new value in an asynchronous way. S_2 has therefore a good, but not necessarily *correct* or *latest* value of x.

Each occurrence of a tilded variable in the program text does not lead necessarily to a different asynchronous variable. Since asynchronous variables are associated to processes, all occurrences of the tilded variable $\tilde{x}$ in a given process P_i are identical.

Definition 3 *Let $\widetilde{Var}(instr)$ be the set of tilded (global) variables used and defined in instr, $\widetilde{Var}(I) = \cup_{i \in I} \widetilde{Var}(i)$.*

The definition of $\widetilde{Var}$ allows us to uniquely identify the different occurrences of $\tilde{x}$: simply replace all occurrences of $\tilde{x}$ in $\widetilde{Var}(P_i)$ with $\tilde{x}_i$.

In the next section we will informally detail the semantics of $_{\mathrm{d}}$SL, without going into the description of well known programming concepts such as expression evaluation, method-call, and control flow. We only emphasize on $_{\mathrm{d}}$SL's special features : its processes behavior, the processing of whens and messages, the behavior of assignments, and finally its sequences.

Process behavior

The behavior of a $_{\mathrm{d}}$SL program in conjunction with the environment can be seen as

$$S_1 \parallel \cdots \parallel S_k \parallel \mathcal{E}$$

where each S_i is an independent process that governs the variables in P_i and $\mathcal{E}$ is the environment. We associate to each process S_i a fifo queue F_i that models the communications between the different sites and is defined as follows.

F_i is tuple $< A, f, b >$ where $f, b \in \mathbb{N}$ and A is a function $\mathbb{N} \mapsto \mathcal{D}$. Four operations are defined on F_i :

- $First(F_i) :< A, f, b > \mapsto A(f)$ if $f < b$, $\perp$ otherwise
- $EnQueue(F_i, d \in \mathcal{D}) :< A, f, b > \mapsto < A[A(b) \mapsto d], f, b+1 >$
- $DeQueue(F_i) :< A, f, b > \mapsto < A, f+1, b >$
- $Size(F_i) :< A, f, b > \mapsto b - f$

$\mathcal{D}$ describes the domain of the messages in F_i, and is defined as the set of messages that update the value of an asynchronous variable and the request to execute a particular portion of code. $\mathcal{D} = \mathcal{D}_{Tilde} \cup \mathcal{D}_{Launch}$. To update a variable, the message contains the designated variable, and its latest value : $\mathcal{D}_{Tilde} = Var \times \mathbb{Z}$. To execute some sequential code, a process receives a pointer to the code that has to be executed. $\mathcal{D}_{Launch} = Lbl$, where Lbl is the set of possible labels for sequential code. Remark that using these primitives models a reliable, but not necessarily instantaneous, communication channel.

Execution of each S_i is an infinite loop that executes the code in figure 3, each iteration of the loop is called in cycle, or more precisely an *Input-Process-Output* cycle because it contains tree phases. (1) Input : variables linked to inputs change their value according the physical state of the device they are attached to, (2) Process : events are triggered and incoming messages are processed (3) Output : variables linked to outputs force the physical state of the devices they are connected to.

// **Input**
  `for each input variable` $x \in P_i$, **do** $x :=^{\Delta} x_*$ **od** [4]
// **Process whens**
  `for each` $W_j : Var(W_j) \cap Var(P_i) \neq \phi$ **do** `execute` W_j **od**
// **Process messages**
  `take` n `in` $0, \cdots, Size(F_i)$ [5], `for each` $j = 0, \cdots, n-1$, **do**
    $msg \in \mathcal{D} = First(F_i); F_i \mapsto DeQueue(F_i); Handle(msg);$ **od**
// **Output**
  `for each output variable` $x \in P_i$, **do** $x_* :=^{\Delta} x$ **od** [4]

Figure 3: Input-Process-Output behavior

[4] where x_* is the hardware value of the variable x and $:=^{\Delta}$ is the usual definition of assignment (ie. without whens), cfr subsection on assignment.

[5] Some results in section 6 use instantaneous message passing, which forces $n = Size(F_i)$. Remark that choosing infinitely often 0 models communication breakdown.

Processing whens

To each when W_i of the form `when` *Cond* `then` *Body* `end_when`, we associate a hidden variable W_i'. The pseudo code for the execution of W_i is

if $Cond \wedge \neg W_i'$ **then** $W_i' := true$; *Body* **else** $W_i' := Cond$ **fi**.

Remark that different execution orders of a given set of whens may lead to different results. To cope with this problem, each site processes its whens respecting their order of appearance in the program text.

Processing messages

The processing of messages is straightforward, if a message $(x, val) \in \mathcal{D}$ is received in S_i, it executes $\tilde{x}_i := val$ (cfr section on assignment). A message containing a label will cause the receiving process S_i to execute the code associated to that label, until it reaches the end of that code.

Assignment

An assignment of the form `x := e;` executed on site S_i has the usual result (x is initialized with the evaluation of e), but $_d$SL adds two features to the assignment. First of all, all whens W_j are executed such that $x \in Var(W_j) \cap Var(P_i)$. Secondly, if `x` has an asynchronous distant copy, then the asynchronous variable(s) is (are) updated. $\forall j : \tilde{x}_j \in P_j$, do $F_j \mapsto EnQueue(F_j, (x, e))$

Remark that the special behavior for assignment may cause infinite recursion in the processing of whens. A simple static check[6] allows us to reject programs that may contain unacceptable recursion.This allows to model the assignment `x:= e;` by a common assignment followed by an inlining of all whens that have `x` in their condition.

Sequences

Sequences are containers for independent sequential code. Because sequences allow atomic executions, they allow to execute consecutive instructions without the locality constraint imposed on event-driven code. This means that they can contain a sequence of instructions that use and define variables governed by different S_i. However, an instruction in a sequence is actually executed by a particular S_i depending on the variables it manipulates. Imagine a label on each instruction in each sequence, and S_i executes $instr_1$ on lbl_1, while $instr_2$ is the next instruction with lbl_2. If $Var(instr_2) \cap Var(P_i) = \phi$, then S_i stops the execution of the sequence, and $F_j \mapsto EnQueue(F_j, lbl_2)$ for $j : Var(P_j) \cap Var(instr_2) \neq \phi$.

Note that $Var()$ contains only global variables which implies that local variables do not influence the localization of instructions in sequences. It is the implementation of sequences that makes sure that these local values are correctly communicated between all intervening sites. We suppose here that all instructions that can be localized ($\forall instr : Var(instr) \cap \cup_j Var(P_j) \neq \phi$).

4 Case study : a canal lock controller

To illustrate $_d$SL concepts, we study the design of a controller for a system composed of two consecutive locks. Each lock is composed of two gates, a top and a bottom one. In between the top and the bottom gates of each lock, the water level can be controlled (i.e. the inside of a lock can be filled of emptied). The different commands of this system (opening/closing a gate, emptying/filling a lock) can be accessed via a control panel. For this system to function properly, several constraints must be satisfied: (1) two consecutive gates cannot be opened at the same time, (2) a gate can only be opened if the water level on each side is the same, and (3) the water level inside a lock can only be changed if both its top and bottom gates are close. The purpose

[6]a sufficient condition is that for each W_i, $Instr(W_i)$ contains at most one assignment to each variable that appears in the condition of that W_i.

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 idea to implement the controller in $_d$SL 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 $_d$SL source can be found in appendix B).

```
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;
```

Figure 4: `when` monitoring the command "open the bottom gate of lock2"

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 delay, an order might still 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.

5 $_d$SL to *Promela*

The application

The semantics of $_d$SL allows an almost immediate translation of a given $_d$SL program to *Promela*. We show how a real controller application is translated from $_d$SL into *Promela*, and how, with the help of *Spin*, some errors in an initial design were discovered.

Consider the $_d$SL program in appendix B, which reveals eleven independent processes, based on the maximal partition where one element contains $\cup_{i\in\{7,8,9,10,13,14,15,16\}} Instr(W_i)$, and each of the other ten elements contains the instructions of one of the ten remaining whens. This is the maximum number of sites on which the program can be distributed. However, the actual distribution can use less sites, by forcing some variables on the same site. Here, the physical distribution uses 3 sites, (e.g. close buttons are part of the site that also governs the open buttons) : S_1, S_2, S_3, that govern respectively the variables

- `lock1.{top/bottom}_gate.motor_{command/direction}`,
  `lock1.water_{command/direction/up/down}`,
  `lock1.{top/bottom}_gate.{opened/closed}`.
- `lock2.{top/bottom}_gate.motor_{command/direction}`,
  `lock2.water_{command/direction/up/down}`,
  `lock2.{top/bottom}_gate.{opened/closed}`.

- `lock1.{top/bottom}_gate.button_{open/close}`,
  `lock2.{top/bottom}_gate.button_{open/close}`,
  `lock1.button_{empty/fill}`,
  `lock2.button_{empty/fill}`,
  `lock1.{top/bottom}_gate.order_given`,
  `lock2.{top/bottom}_gate.order_given`.

Remark *Since the output variable* `not_allowed_led` *has no influence on the behavior of the system, we removed it from the application.*

Modeling the environment

If one considers the environment as an individual process that reacts on outputs and continually changes the inputs of the dSL program, the result is a surprisingly large state space that is much too big for verification purposes. To cope with this problem, consider the behavior of a process S_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. To avoid unnecessary interleavings between the environment and the different dSL processes, the part of the environment that is connected to S_i is *frozen* as long as S_i is in its process phase. For this particular application, this simplification is legitimate since we can split the environment up into three independent parts where each part interacts with only one S_i. To simplify the model, the environment is integrated into the specification of each S_i (by means of *inlining*), and allowed to change state when the process reaches its input phase.

In our application, the environment contains four gates, two water levels and an operator that can press twelve buttons. The gates and the water levels are modeled using a *flip-flop* behavior (cfr figure 5) that has four states (with the corresponding bits for sensor_flipped and sensor_flopped) : flipped (1,0), flopped(0,1), flipping(0,0) and flopping(0,0). It receives 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 between the twelve buttons (`{lock1/lock2}.{top/bottom}_gate.button_{open/close}` and `{lock1/lock2}.button_{empty/fill}`).

```
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;
      :: ! order_cmd -> skip;
   fi;
}
```

Figure 5: Flip flop behavior

Modeling the Processes S_i

The processes S_i described in the semantics can be coded almost as-is in *Promela* using a set of macros interpreted by the C precompiler. To reduce the size of the state space, note that all actions taking place in the process phase of a given process are deterministic ($_d$SL does not allow non-determinism, by imposing an ordering on the processing of the whens for example), and intermediate states have no effect on the behavior of the other processes. In other words, the process phase, which contains most of the controller's behavior, is a deterministic sequence of internal states. Using the `d_step` feature from *Promela* allows to merge all such trails into single nodes in the state space. Unfortunately, when the $_d$SL semantics is modeled in *Promela* the limited size of the queues associated to each process make the `d_step` illegal when a message is sent to a full queue. Replacing it with `atomic` is a less but still good alternative that yields a spectacular reduction of the state space by not allowing any interleaving of `atomic` executions of different processes (cfr fig 6).

Communications between the different processes are modeled using *Promela*'s `chan` and are kept reliable but not instantaneous.

```
chan ch_1 = [MAX_CHANNEL_SIZE] of { byte, int, int };
active proctype VM_1 () {
  atomic { init_1(); }
  do::
    // Read Inputs
    atomic { input_1(); }
    // Handle whens for changed Inputs)
    atomic { when_1();  }
    // Read messages
    atomic { read_msg_1(); }
  od;
}
inline input_1() {
  // Environment for lock1.top_gate
  flip_flop_behavior (
   lock1_top_gate_opened, lock1_top_gate_closed,
   lock1_top_gate_motor_command, lock1_top_gate_motor_direction);
  ... // Same for lock1.bottom_gate, and lock1.water
}
inline when_1() {
  // Processing for WHEN lock1.top_gate.closed || lock1.top_gate.opened THEN
  //                     lock1.top_gate.moter_command := false
  //                END_WHEN
   if :: !(lock1_top_gate_closed || lock1_top_gate_opened)
         || _old_cond ->
         _old_cond = lock1_top_gate_closed || lock1_top_gate_opened;
      :: else ->
         _old_cond = true;
         lock1_top_gate_moter_command = false;
   fi;
  ... // Same for the other whens on site 1
}
inline read_msg_1 () {
  byte msg_id; int par1, par2;
  do
    :: ch_1 ? msg_id, par1, par2;
    // Dispatch the message
       if :: msg_id == VAR_CHANGED ->
     // The message contains an update for the variable ~x1
             if par1 == ID_x -> x = par2;
             ...
          :: msg_id == REXEC ->
     // The message contains a label that must start sequence1 at lbl1
             if par1 == lbl1 -> start_sequence_1_lbl1();
             ...
       fi;
    :: skip-> break;
  od;
}
```

Figure 6: Model for S_1

6 Results

Problem in the locks controller!

At first glance, the implementation presented in section 4 seems to works. However, after modeling it in *Promela* as explained in section 5 and using *Spin* model checker to verify the given constraints, we found out that it is faulty. Indeed, as shown in figure 7, two consecutive gates (top gate of lock 1 and bottom gate of lock2) 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 '~'). So 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, so it allows the opening of the bottom gate of lock 2, which violates the constraints.

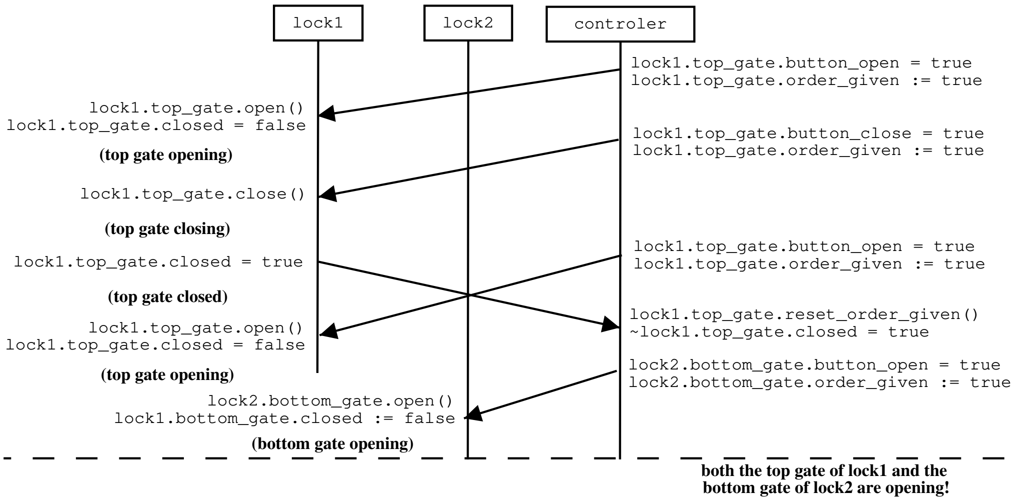

Figure 7: Error trace

An easy way to correct that, 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 constraints. To achieve this, a sequential execution checks that the issued command can be executed, by migrating the condition to all sites intervening sites. As illustrated in figure 8, 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 8, 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

The constraints expressed in section 4 are expressed in LTL as a safety property using the formula `[] !bad` and checked with *spin* using

```
#define bad ((!lock1_top_gate_closed && !lock1_bottom_gate_closed)
```

```
when lock2.bottom_gate.button_open and not disabled then
  disabled := true;
  launch open_bottom_gate_lock2;
end_when

sequence open_bottom_gate_lock2
begin_var
  check : bool;
end_var
  check := (lock2.top_gate.closed and lock2.water_down);
  check := (check and lock1.top_gate.closed);
  if check then
    not_allowed_led := false;
    launch lock2.bottom_gate<-open();
  else
    not_allowed_led := true;
  end_if;
  disabled := false;
end_sequence
```

Figure 8: `when` / `sequence` monitoring the command "open the bottom gate of lock2"

```
|| (!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))
```

Model	Chan Size	Delay	Buttons	Verified	Time	States	Memory (MB)
1	5	No	8	Yes	0:1	$7\ 10^3$	0.6
1	10	No	8	Yes	0:1	$7\ 10^3$	0.6
1	10	No	12	Yes	0:4	$8\ 10^3$	3.4
1	5	Yes	8	Yes	0:1	$12\ 10^3$	0.8
1	10	Yes	8	Yes	0:1	$12\ 10^3$	0.8
1	10	Yes	12	Yes	0:11	$144\ 10^3$	5.7
1	10	No	2x8	**No!**	0:13	$250\ 10^3$	8
1	10	Yes	2x8	**No!**	0:52	$673\ 10^3$	23
1	10	No	2x12	**No!**	0:26	$524\ 10^3$	18
1	10	Yes	2x12	**No!**	1:23	$1.14\ 10^6$	39
2	5	No	8	Yes	0:1	$8\ 10^3$	0.6
2	10	No	8	Yes	0:1	$8\ 10^3$	0.6
2	10	No	12	Yes	0:9	$286\ 10^3$	8.6
2	5	Yes	8	Yes	0:1	$8\ 10^3$	0.6
2	10	Yes	8	Yes	0:1	$8\ 10^3$	0.6
2	10	Yes	12	Yes	0:9	$290\ 10^3$	8.8
2	10	No	2x8	Yes	1:22	$3.14\ 10^6$	89
2	10	Yes	2x8	Yes	1:54	$3.46\ 10^6$	98
2	10	Yes	2x12	-	6:41:00	$+1.7\ 10^8$	+1500

Figure 9: Results

Results for the actual verification with the simplifications described before, can be found in figure 9. The first column indicates which model was taken, followed by the number of messages the channels can hold, a flag that indicates whether messages are emptied as soon as possible from their queues, next the number of buttons the operator can press. The last four columns show whether the safety constraints were verified, the time it took *Spin* to do so, and the memory it needed expressed in number of states and actual memory.

Closer inspection of the results in figure 9 reveal first of all that the size of the channels used for message passing has no influence on the size of the state space. Secondly, remark that the use of *instantaneous* message passing (a process never executes a cycle without emptying its queue) has an impact on the state space of model 1 but makes almost no difference for model 2. This is due to the fact that no *tilded* variables are used in model 2, and because the model does not *progress* until the *launch* messages are treated. A site that executes a cycle without reading such a message creates almost no *new* states in the state space.

More critical information is contained in the number of buttons the operator may press on. Since the problem is completely symmetric, the constraints may be checked on a part of the system that does not contain the water level and the outermost gate of one of the locks. In that case, four buttons can be disabled : open/close for the omitted gate and up/down for the water in the lock. Note that, to find the error, one must allow the operator to press more than once on each button (indicated in figure 9 by 2x). Unfortunately, to keep the verification acceptable both in time and memory, the operator is restricted to press each button no more than twice. Even with such a restrictive behavior, we were unable to verify the entire second model, and had to verify the constraints on the 2x8 version. The reason for the explosion of the state space in model 2 is the absence of restrictions on the behavior of the process that controls the control panel. In model 1, pressing a button does not necessarily trigger the event since the conditions on the *tilded* variables must be satisfied. In model 2, only the disabled variable can stop an event from being triggered when a button is pressed. Finally, note that the results revealing the violation of the property are hard to interpret in relation to the other results, since the verifier did not have to explore the complete state space.

7 Conclusions

In this paper, we presented ${}_{\mathrm{d}}$SL, a distributed environment for designing industrial process control. We pointed out the main advantages of using ${}_{\mathrm{d}}$SL. This is achieved by offering a transparent code distribution that simplifies the designer's task. After briefly presenting ${}_{\mathrm{d}}$SL semantics, which is based on the notion of *maximal distributability*, we showed how a ${}_{\mathrm{d}}$SL program can be modeled in *Promela*. This allowed us, using the *Spin* model checker, to detect some non trivial flaws in the initial design of a locks controller. Finally, we commented some verification results, and showed that this technique can not be applied as is : great efforts to reduce state space are required in order to obtain a verifiable model. In the future, we would like to automate the entire translation of ${}_{\mathrm{d}}$SL 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 a 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 ${}_{\mathrm{d}}$SL library of classical distributed mechanisms (i.e. semaphores, mutex).

References

[AO91] Krzysztof R. Apt and Ernst-Rudiger Olderog. *Verification of Sequential and Concurrent Systems.* Texts and Monographs in Computer Science. Springer-Verlag, Berlin, New York, 1991.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Inc., Reading, Mass., 1986.

[DW03] Bram De Wachter. Code Distribution in the dsl Environment for the Synthesis of Industrial Process Control. Technical report, U.L.B., 15 January 2003.

[EC99] D.A. Peled E.M. Clarke, O. Grumberg. *Model Checking.* MIT Press, 1999.

[GMM+01] F. Geurts, F. Macq, T. Massart, A. Piron, and G. Vanstraelen. Présentation du projet distributed sl (${}_{\mathrm{d}}$SL). Technical Report 444, ULB, 2001.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[Hol97] G. J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[Mil89] R. Milner. *Communication and concurrency.* Prentice-Hall, Inc., 1989.

[PW01] A. Piron and B. De Wachter. The ${}_{\mathrm{d}}$SL language proposition : Grammar. Technical Report XX1, ULB, 2001.

A Grammar

Remark *This grammar is a reduced version of the complete ${}_{\mathrm{d}}$SL grammar, and to keep it compact, some rules may introduce ambiguity.*

Dsl_Program ⟶ *dsl_element_list init*

dsl_element_list ⟶ *dsl_element dsl_element_list* | *dsl_element*

dsl_element ⟶ *global_var_declaration* | *method_declaration* | *when* | *sequence*

init ⟶ “**program**” “**begin_var**” *id_list* “**end_var**” *instruction_list* “**end_program**”

global_var_-declaration ⟶ “**global_var**” *var_list* “**end_var**”

var_list ⟶ *id_list* “**:**” *var_type* | *id_list* “**:**” *var_type* “**;**” *var_list*

id_list ⟶ *id* | *id* “**,**” *id_list*

var_type ⟶ “**int**”, “**input_byte**”, “**output_byte**”

method_declaration ⟶ “**begin_method**” *id* “**(**” *id_list* “**)**” “**begin_var**” *id_list* “**end_var**” *instruction_list* “**end_method**”

instruction_list ⟶ *instruction instruction_list* | *instruction*

instruction ⟶ *if* | *while* | *assign* “**;**” | *synchronous_method_call* “**;**” | *asynchronous_method_call* “**;**” | *end_if* “**;**” | *sequence_call* “**;**” |

if ⟶ “**if**” *expr* “**then**” *instruction_list* “**else**” *instruction_list* “**end_if**”

while ⟶ “**while**” *expr* “**do**” *instruction_list* “**end_while**”

assign ⟶ *id* “**:=**” *expr*

synchronous-_method_call ⟶ *id* “**(**” *id_list* “**)**”

when ⟶ “**when**” *expr* “**then**” *instruction_list* “**end_when**”

sequence ⟶ “**sequence**” *id* “**begin_var**” *id_list* “**end_var**” *instruction_list* “**end_sequence**”

asynchronous-_method_call ⟶ “**launch**” *synchronous_method_call*

sequence_call ⟶ “**launch**” *id* “**(**” *id_list* “**)**”

expr ⟶ *expr bin_op expr* | *un_op expr* | “**(**” *expr* “**)**” | *id* | *cst*

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

un_op ⟶ “**-**” | “**not**”

B Lock controller : dSL source

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 *)
PROGRAM LOCK
  not_allowed_led := FALSE;
END_PROGRAM
```

Second attempt

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

METHOD GATE::open(), METHOD GATE::close(), METHOD GATE::reset_order_given()
WHEN IN GATE self.closed OR self.opened, METHOD LOCK::empty(), METHOD LOCK::fill()

(* 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. *)
...

PROGRAM LOCK
  not_allowed_led := FALSE;
END_PROGRAM
```