$_{\text{d}}$SL: An environment with automatic code distribution for industrial control systems

Bram De Wachter[⋆], Thierry Massart, and Cédric Meuter[⋆]

University of Brussels (ULB)
Département d'Informatique, Bld du Triomphe, B-1050 Bruxelles
{bdewacht,tmassart,cmeuter}@ulb.ac.be

Abstract. We present and motivate the definition and use of the language and environment $_{\text{d}}$SL, an imperative and event driven language designed to program distributed industrial control systems. $_{\text{d}}$SL provides transparent code distribution using simple mechanisms. Its use allows the industrial control system's designer to concentrate on the sequences of control required; the $_{\text{d}}$SL compiler-distributer taking into account the distribution aspects. We show the advantages of our approach compared to others proposed using e.g. shared memory or synchronous languages like Esterel, Lustre or Signal.
keywords: Industrial process control, transparent code distribution, execution migration

1 Introduction

An industrial control system is generally safety critical, event-driven, physically distributed and controls heterogeneous equipments whose response time can range from milliseconds to minutes. To be of any use in a real industrial environment, a control system must be reliable, efficient, robust and simple. Efficiency is needed to ensure that the controller is not overtaken by the system it controls. Robustness allows a maximal control even in case of hardware failure (sensor, actuator or processor). Simplicity is of main importance to allow a strong monitoring of the system and in case of maintenance or upgrade, to be able to easily update it without stopping the industrial system controlled.

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, it is beneficial to design a development environment which handles the communication aspects and allows the programmer to concentrate on the functional aspects of the system. Classical solutions based on this idea exist (CORBA[16], DCOM [22], EJB [10]). Unfortunately, due to the genericness of these solutions, they are quite heavy and completely hide all of the communication process, making the monitoring of such systems difficult.

More dedicated solutions to the problem of distributed execution of a system with transparent distribution mechanism, have been proposed. Examples of such solutions are distributed shared memory [27], or more specifically in the domain of control systems, synchronous languages like Esterel, Lustre or Signal (e.g. [2] and [15]).

⋆ Work supported by the *Region de Bruxelles Capitale*, grant no. RBC-BR 227/3298.

Unfortunately, even if shared memory solutions are generally lighter than the distributed objects one, due to the cache coherence protocol, the time to access the memory can vary greatly and is not predictable. We also motivate why in our opinion, the latter solutions have, in practice, some drawbacks.

This leads us to the definition of $_{\mathrm{d}}$SL[1], a new environment and language designed to program distributed industrial control systems, providing transparent code distribution using low level mechanisms adapted for the industrial environments. $_{\mathrm{d}}$SL has been developed by the verification group of ULB[2] in collaboration with the company Macq Electronique[3]

$_{\mathrm{d}}$SL offers both advantages to allow, most of the time, $_{\mathrm{d}}$SL programmers to ignore all the communication aspects between controllers of the distributed systems and, by the simplicity of the distribution mechanisms, to easily monitor the behavior of the synthesized distributed system. $_{\mathrm{d}}$SL can also be formally modeled and therefore allows links with the world of formal model-checking to verify the correctness of the systems. Another advantage of this approach is the ability to debug and verify the centralized program before its distribution.

In the remaining part of this paper, we first, in section 2, detail related proposals and justify the advantages of our solutions. In section 3 we present the $_{\mathrm{d}}$SL syntax and in section 4, outline its semantics. In section 5, we describe the distribution procedure and $_{\mathrm{d}}$SL environment. Finally in section 6, we discuss our future work.

2 Other approaches and motivations

The problem of distributing applications that control reactive systems has been studied for many years now and several interesting observations on these works shaped the design of $_{\mathrm{d}}$SL. In particular, this problem has been studied conceptually in the world of process algebra and defined as a correctness preserving transformation of a *centralized* specification into a semantically equivalent distributed one. (e.g. for bisimulation equivalence [24], see [21, 7]). It has also been studied on various types of labelled transition systems ([9] [26, 28]).

These works solved part of the problem. However, contrary to other programming languages, the notion of variable does not exist in process algebra and these solutions had therefore to be extended. Work has also been done in the domain of synchronous languages such as Esterel [6], Lustre [8] and Signal [20], which answered questions on how to specify controllers in a natural and semantically well defined way. Unfortunately, in our opinion, the distribution of synchronous languages while preserving the semantics, suffer from a performance problem which, in practice, may not be acceptable.

Indeed, the synchronous programming scheme found in the synchronous languages supposes that time is defined as a sequence of *instants*. To preserve determinism, these languages use the concept of synchronous broadcast [5, 4] when several processes are composed in parallel. This implies that parallel branches in the high level description can be transformed into sequential deterministic code. The distribution of such programs, for example in Esterel [15], may suffer from

[1] $_{\mathrm{d}}$SL is the successor of the language SL (Supervision Language, based on *ST*, an industrial standard defined in IEC1131-3) developed at Macq Electronique company

[2] http://www.ulb.ac.be/di/ssd/groupverif.html

[3] a leading company in industrial process control, http://www.macqel.be

severe performance penalties 1. because the *instants* must be respected, requiring a strong resynchronization scheme, and 2. because the distribution is applied on the determinized sequential code. The distribution of Esterel described in [15] can be summarized in 4 steps : (1) the *centralized* program (after being compiled in a single threaded sequential code) is duplicated on all participating sites; (2) the instructions that are not relevant to a given site are removed; (3) for data that is accessed on one site, but calculated on another, communication messages are inserted, and (4) synchronization messages are inserted to preserve the global instants. Remark that since the initial code is sequential, this solution suffers from the lack of parallelism (there are some ways to achieve higher concurrency such as weak synchronization but that does not preserve safety properties [15]). For Signal, the situation is very similar [2].

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

These observations make us believe that, although perfectly suitable for tightly coupled homogeneous systems and having the benefit of simplicity when it comes to specifying a controller, the simplicity of the synchronous approach is too costly in terms of performance when applied to loosely coupled heterogeneous systems. Moreover, from the experience of our industrial partner specialized in process control, the strong synchronization of all processes is only rarely needed and must therefore not be used by default, but made available when needed.

To avoid these drawbacks, $_{\rm d}$SL rejects the synchronous product [23] used in the above languages at the detriment of indeterminism, and adopts an asynchronous composition of instantaneous (*atomic*) code and asynchronous (*sequential*) code. Asynchronous composition is therefore the keyword in $_{\rm d}$SL's design. The instantaneous 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. Moreover, as we detail later on, $_{\rm d}$SL offers a way to detect and handle network or hardware failures. 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, as explained in section 5.

For the distributed execution of the *sequential* code, several models are proposed in the literature. These models can be divided into two sets based on the way they achieve data locality : either move data, or move the execution. Many systems have been studied that use the first solution, such as Distributed Shared Memory systems [27]. These systems, although offering a transparent distributed environment, suffer from undesirable border effects that make them unusable in an industrial environment. The need to replicate data to make such systems work in a performance responsible way [17], may cause thrashing[4] or false sharing[5]

[4] The effect of two (or more) processes competing for exclusive access to a given variable, resulting in high communication traffic and almost no productiveness

[5] Caused when two (or more) variables, used by different processes, are in the same page causing unnecessary communication traffic

and these systems therefore do not guarantee stable performance as observed in [27]. Secondly, since data moves around, the supervision of such systems and its error-recovery - both indispensable features in industrial applications - may become too complicated [25] on dSL's target hardware.

For these reasons, dSL uses the second solution, which consists of moving the execution to the data, a concept known as process or thread migration [14]. In this concept, a thread of execution is halted on one site, its context (local variables and program counter) is sent to another site, where its context is restored and execution continues. Thread migration is known to enable dynamic load distribution, fault tolerance, eased system administration, data access locality and mobile computing [19]. In our system, all instructions and global variables are statically assigned to the participating sites and thread migration, decided at compile time, is used to obtain data access locality. The benefits are twofold: (1) following the state of the system is very easy, and (2) all communication and synchronization messages can statically be calculated, resulting in a predictable execution. However, we lose the benefits of dynamic load balancing and fault tolerance since the migration policy used in dSL is static.

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

3 The dSL concept

dSL is an imperative language with static variables. Each variable can be either (1) internal to a program, (2) linked to an input (sensor), or (3) linked to an output (actuator). dSL is *event driven.* This allows to specify that when the value of a boolean expression switches from false to true, some code must be executed. For instance, `when x >= 0 then run_motor1(); end_when` will trigger the method `run_motor1()` every time the variable `x` switches from a negative to a positive value. dSL also offers limited *Object Oriented* features.

Moreover, the domains of all dSL primitive types are extended with the special value `unknown`. A variable linked to an I/O may take this special value in case of hardware failure. The `unknown` value propagates in expression evaluation and can be tested for with the builtin `is_unknown` statement. As shown in figure 1 this allows to construct more robust programs. In this figure, a sensor is duplicated in order to ensure correct behavior in case of hardware failure. Note that the body of a `when` whose condition evaluates to `unknown` is not executed.

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

Fig. 1. Fault tolerance in dSL with unknown.

A program in dSL is written in a centralized manner, as if every input or output can be accessed without the need for explicit communication or synchronization (we shall see that some restrictions are imposed to apply this principle).

The designer must then fill in a *localization table* to specify the physical localization (execution sites) of each I/O. Other (internal) variables are either global in which case their localization will statically be fixed by the distributer, or local in which case they can move during execution. Since global variables do not move during execution, the distributer has to ensure that an instruction accessing a global variable is executed on the site of that variable. An execution site can be either a supervisor (typically a computer, possibly with a user interface) or a programmable controller (called automata from here on, which are connected to the industrial equipment through the sensors and actuators).

The $_{\mathrm{d}}$SL compiler/distributer automatically distributes the code among the execution sites, trying to minimize communications, and compiles the distributed code to an assembler-like language. This assembler-like code is interpreted by a $_{\mathrm{d}}$SL *Virtual Machine*. A $_{\mathrm{d}}$SL virtual machine is available for both supervisors and automata. This is illustrated in figure 2.

This approach has many benefits such as (1) maintainability (only one language is used to program both the supervisors and automata) (2) flexibility (any change of an actuator or a sensor does not imply changes in 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).

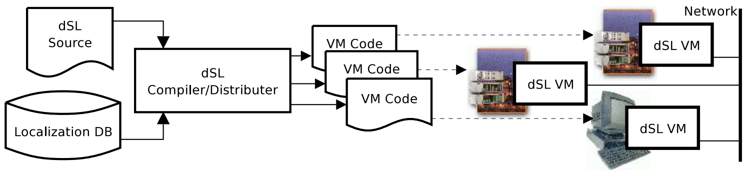

Fig. 2. $_{\mathrm{d}}$SL

A $_{\mathrm{d}}$SL program contains several parts. (1) class declarations, (2) global variables declarations - including all I/O variables (3) method definitions (4) when definitions (5) sequence definitions and (6) a program initialization. Each Input (resp. Output) variable v_{in} (resp. v_{out}) is linked to a hardware sensor v^*_{in} (resp. actuator v^*_{out}). For A simplified version of the $_{\mathrm{d}}$SL grammar, see appendix A.

Atomic and sequential code. The design of $_{\mathrm{d}}$SL has been dictated by the execution paradigm requiring an immediate reaction to events and their instantaneous treatment. In practice this forbids any implicit synchronization during the execution 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, in $_{\mathrm{d}}$SL, there is a distinction between:

- *atomic* code which must be executed in an atomic manner and therefore cannot be distributed,
- *sequential* code which can be distributed and use inter-site communications to synchronize or transfer values between sites.

```
CLASS Heater
    control, state : INT;
    maintenance    : BOOL;
END_CLASS

GLOBAL_VAR
    heater                : Heater;
    temperature, fuel_cost : INT;
    alarm, led            : BOOL;
END_VAR

SEQUENCE set_heater(new_state : INT)
    heater.control := heater.control + 1;
    heater.state := new_state;
    IF (heater.state == 1) THEN
        led       := TRUE;
        fuel_cost := fuel_cost + 10;
    ELSE
        led := FALSE;
    END_IF
END_SEQUENCE

WHEN IN Heater (control==1000) THEN // W1
    control := 0;
    maintenance := TRUE;
END_WHEN
```

```
WHEN heater.maintenance THEN        // W2
    alarm := TRUE;
END_WHEN

WHEN ~temperature < 0 THEN          // W3
    IF (NOT heater.maintenance) THEN
        LAUNCH set_heater(1);
    END_IF
END_WHEN

WHEN ~temperature > 20 THEN         // W4
    IF (NOT heater.maintenance) THEN
        LAUNCH set_heater(0);
    END_IF
END_WHEN

PROGRAM
    heater.control := 0;
    heater.maintenance := FALSE;
    LAUNCH set_heater(temperature<0);
END_PROGRAM
```

Fig. 3. A temperature control system in dSL

Code inside a WHEN (the instruction inside its body and the condition) is forced *atomic*, and must therefore be local to a given site. *Sequential* code is defined through the use of the SEQUENCE construct. The code inside a METHOD can be either *atomic* or *sequential* depending on the context in which it is called. If a METHOD can be reached from a WHEN, then the body of this METHOD is assumed to be atomic. It is assumed *sequential* otherwise. To relax the *atomic* constraints in a WHEN, two mechanisms have been defined (see figure 3) :

– The LAUNCH keyword allowing to call a SEQUENCE or a METHOD asynchronously (i.e without waiting for the control to return from the SEQUENCE or the METHOD), and possibly on a distant site. Note that a SEQUENCE can only be called asynchronously (using LAUNCH) and that it cannot have more than one instance executed simultaneously.
– The “~” operator allowing to reference the last locally known value of a variable possibly on a distant site. When the value of a variable is changed on the site governing it, its new value is sent to all necessary sites. One must be careful with *tilded* variables since it is never guaranteed that the value of the *tilded* variables corresponds to the real value of the variable. It can be interesting to use if the exact value is not imperative (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 (e.g. using a procedure for explicit synchronization). A site that has a tilded copy of x, regularly checks if the site owning x is still alive. If not, the copy is set to unknown, indicating hardware or network failure.

dSL example. To illustrate the dSL concepts, let us examine a small example of a temperature control system. In this system, a temperature sensor is linked to an input variable temperature. A heater is turned on (off) if the temperature is below 0^o (above 20^o). The state of the heater (on/off) is controlled by the output variable heater.state. Moreover, there are two indicators on a control panel. The first indicator, (linked to the output variable led) is used to indicate the state of the heater, and the second (linked to the input variable `alarm`) is updated when the heater has been turned on a certain number of times. An additional variable `fuel_cost` estimates the amount of fuel consumed by the heater. The $_{\text{d}}$SL program is presented in figure 3.

4 The $_{\text{d}}$SL semantics

In this section, we introduce the $_{\text{d}}$SL semantics, concentrating on the distributed aspect of the language. We therefore skip a complete and formal review of well known program issues like method call, control flow, expression evaluation and the limited object oriented features.

The behavior of a $_{\text{d}}$SL program depends on the localization of its variables. Our goal is to describe the semantics of a $_{\text{d}}$SL program independently from any localization information. For that, we introduce the notion of *maximal distribution*, which expresses the most permissive way to distribute a $_{\text{d}}$SL program. The semantics of a $_{\text{d}}$SL program is then defined by the set of all behaviors of its *maximal distribution.*

Maximal distribution. The maximal distribution is deduced from the locality constraints imposed on global variables by the *atomic* code, e.g (1) two global variables appearing in the same instruction and (2) two global variables accessed by the same `WHEN` must be governed by the same site. This defines a partition of the set of variables where each subset of the partition corresponds to an execution site. A formal description of how to find the maximal distribution can be found in appendix B.

Process behavior. The behavior of a $_{\text{d}}$SL program P in its *maximal distribution* is given by

$$P_1 \parallel ... \parallel P_n$$

where each P_i is an independent process executing the part of code of P handling all the variables local to site i. These processes communicate through FIFO-channels between each pair of processes. We will note $F_{i,j}$ the FIFO-channels used from a process P_i to another process P_j.

Every process P_i is an infinite loop. Each cycle (i.e iteration) is composed of three phases: (1) the input phase, where each physical input is sampled and where the variables linked to those inputs are updated, (2) the process phase where the necessary `WHEN`s are triggered and where the messages from other execution site are processed and (3) the output phase where the physical outputs are updated according to the variable they are linked to. The pseudo-code for this *input-process-output* cycle is given by :

```
//input phase:
for each $v_{in} \in P_i$ linked to an input do $v_{in} \leftarrow v^*_{in}$ done
//process phase:
for each $w \in \mathcal{W} \cdot Var_{\mathcal{W}}(w) \subseteq P_i$ do process $w$ done
for each $j \in \{1, 2..., i-1, i+1, ..., n\}$ do process messages from $F_{j,i}$ done
//output phase:
for each $v_{out} \in P_i$ linked to an output do $v^*_{out} \leftarrow v_{out}$ done
```

where v_{in}^* (v_{out}^*) denotes the hardware value of the variable v_{in} (v_{out}), $\mathcal{W}$ the set of `WHEN`s, and $Var_{\mathcal{W}}(w)$ the set of variables accessed by when w (for more details see appendix B).

Processing `WHEN`s. To each `WHEN`, we associate a hidden variable v_w keeping the previous value of the condition. This allows to trigger w of the form "**WHEN**" *cond* "**THEN**" *instruction_list* "**END_WHEN**" only when the condition switched from false to true. Note that the `WHEN`s are processed in their order of appearance in the $_{\mathrm{d}}$SL program. The pseudo code for the execution of w is :

$$\textbf{if } cond \wedge \neg v_w \textbf{ then } v_w \leftarrow \textbf{true}; \text{execute } instruction\_list \textbf{ else } v_w \leftarrow cond \textbf{ fi}$$

Processing Messages. Conceptually, there are two types of messages: (1) messages concerning the update of *tilded* variables and (2) messages concerning the remote execution of *sequential* code. The first kind of message is of the form (v, new_value). Processing such a message simply consists in assigning this *new_value* to the local copy of v (see Assignment hereafter). The second kind of message, corresponding to the `LAUNCH` or continuation of *sequential code*, is simply a *label*. Indeed, when a process must execute remote code, it posts a message with the label corresponding to the first instruction of that code to the governing site. Processing such a message consists of the execution of the code associated with that *label*, until it reaches the end of that code or is migrated to another site.

Assignment. An assignment of the form v "**:=**" e "**;**" executed by the site S_i ($v \in S_i$) has the usual result (the variable v is set to value of the expression e), but $_{\mathrm{d}}$SL adds two features to that. First of all, all `WHEN`s w are processed, as explained previously. Secondly, if v has asynchronous distant copies (i.e. `~v`), then these must be updated. Therefore, for all sites S_j governing a `~v`, a message is posted in $F_{i,j}$ with v and its new value (i.e. the value of e). Note that the special behavior for assignment may cause infinite recursion in the processing of `whens`. A simple static check allows us to reject programs that may contain this unacceptable infinite recursion.

5 Static distribution process of $_{\mathrm{d}}$SL

In this section, we discuss $_{\mathrm{d}}$SL's distribution algorithms and the $_{\mathrm{d}}$SL virtual machine. First, for *atomic* code, the distributer has to assign a unique localization to each instruction and each global variable such that the constraints on *atomic* code are met. Next, for the *sequential* code, the instructions that were not previously dealt with must be localized, taking into account the localization already imposed on the global variables at the level of the distribution of the *atomic* code. A second algorithm is introduced to solve this problem. Finally, we show how, from the computed information on localization, the distribution is actually achieved.

Satisfying the constraints on atomic code. In order to calculate independent components, and to satisfy the localization constraints on *atomic* code, a *dependency graph* is constructed whose purpose is to take into account the dependencies between all instructions and global variables involved in the `WHEN`-part of the $_d$SL program.

Informally, a vertex in this graph is either (1) an instruction that appears or is reachable through synchronous call from the body of any `WHEN` or (2) a non tilded global variable appearing in these instructions. Edges exist between two instructions when control may flow from one instruction to the other, while edges between an instruction and a variable exist if the instruction contains the variable.

Within this graph, an edge states that both vertices must be on the same site to keep *atomic* code local to that site. Each connected component is independent of the others and may be localized on a separate site. If inside one of the components, different sites are specified by the localization table, the program is not distributable. In this case, the designer must relax the *atomic* constraints by introducing `LAUNCH`, “˜” or `SEQUENCE`. Otherwise, if for a given component at least one variable appears in the localization table, all vertices, i.e. instructions and global variables, are localized on the site specified in the table. If this is not the case (not a single variable in the component is present in the localization table), the component can be assigned to any site, e.g. using a load-balancing algorithm.

Remark that this algorithm forces not only the localization of instructions from *atomic* code, but does the same for all instructions in *sequential* code that use global variables also used in *atomic* code. It is up to the algorithm described in the next paragraph to localize the remaining instructions.

Given the following localization table :

`temperature`	Site 2	`heater.state`	Site 2
`alarm`	Site 1	`heater.control`	Not imposed
`led`	Site 1	`heater.maintenance`	Not imposed
`fuel_cost`	Not imposed		

the dependency graph, obtained by applying the previous method on the example of figure 3 is represented in figure 4.a (the vertices W_i correspond to all instructions in W_i of figure 3).

Remark that in this example, all variables but `fuel_cost` are localized once the *atomic* constraints are fulfilled and that imposing `heater.control` on a different site than `heater.maintenance` would make the program not distributable. Also remark that fields from a same class do not necessarily need to be localized on the same site. The ˜ in `W3` and `W4` relaxes the *atomic* constraints, and allows `temperature` and `heater.maintenance` to be localized on different sites.

Localizing remaining sequential instructions. The *sequential* instructions not constrained by the previous algorithm can be localized anywhere. However, it is important to find a good, and if possible, optimal localization. Indeed, as we show further on, between each pair of consecutive instructions of a sequence (control may flow directly from one to the other), localized on different sites, the distributer inserts a migration point so that execution stops on the first site and may continue on the second site. A *bad* localization may result in a program containing unnecessary migration points lowering the performance of the program at runtime.

To evaluate the performance of a particular localization, we introduced the notion of weighted colored control flow graph in [11] : a control flow graph with weights on the edges expressing the mean number of times control will flow following each edge during execution. In the case of an `IF`, these weights are based on the estimated probability of the test being satisfied. For a `WHILE`, they are based on an estimation of the mean number of times the body will be executed. The weights are then obtained by recursively combining these values for nested control structures (e.g. an `IF` branch with probability .3 nested in a loop executing 5 times results in a weight of 1.5). The colors on the vertices model the localization of each instruction (vertices with the same color are localized on the same site). We then define the communication load as the sum of the weights of the edges between vertices of different colors, which corresponds to the mean number of migrations during execution.

The problem of finding the localization minimizing the communication load can be defined as an instance of the NP-complete *Colored Multiterminal Cut* problem which finds the optimal coloring for the uncolored vertices[3]. For a formal definition of the problem and efficient heuristics, which are implemented in our system, see [11].

The figure 4.b illustrates this algorithm based on the example in figure 3 and the results of the previous algorithm. Remark that the algorithm should localize `fuel_cost` on site 1 in order to minimize the communication load.

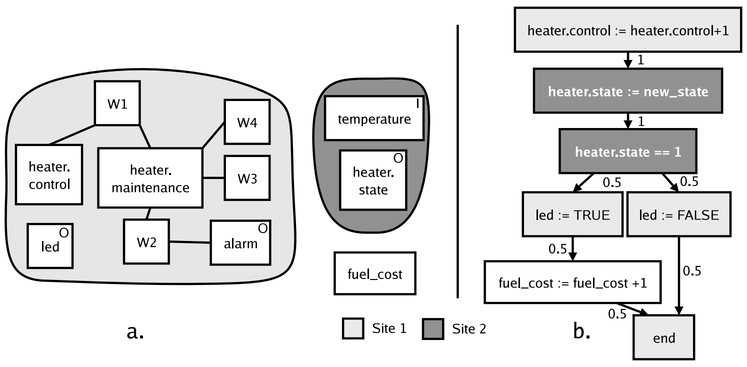

Fig. 4. a. Satisfying constraints on atomic code, b. Localizing sequential instructions.

Executable distributed code. Once every variable and instruction is uniquely assigned to a certain site, the distributer inserts migration points between instructions localized on different sites. The migration of the local context is based on extensive use of *def-use chains* (definition-use, a classical data-flow analysis technique[1]). Since we have complete knowledge of live variables, the distributer can insert code that migrates only those local variables that are updated on the current site and read elsewhere. Technically, context migrating code builds messages to ask the remote update of either register or stack entries on distant sites. A valuable point of our migration method is that in contrast to many systems where the complete stack is migrated [13, 18], $_d$SL can use the information provided by the compiler to migrate only what is needed, saving valuable bandwidth. However, more instructions have to be interpreted to migrate the context than would be needed if the complete stack was migrated. Since in our target platform bandwidth is crucial because network speeds may be of very low quality, our solution yields higher performance.

Execution environment. $_d$SL uses virtual machines. This is clearly indispensable since $_d$SL's heterogeneous target platform consists of servers with a graphical user environment developed for Linux,UNIX,Windows on Intel or Power-PC processors and PLCs (16 bit Motorola 68340 @ 25Mhz with 4MB RAM of which 2MB is flash, no OS) interfaced to the environment. The $_d$SL Virtual Machine, implemented for both server platforms and for the PLC hardware, can be classified as a CISC (i.e. Complex Instruction Set Computing) architecture, interpreting low level three-operand code, is single threaded and has a fixed amount of allocated memory. Simplicity and performance are essential in its design. The instruction set is very rich to optimize the interpretation/execution time ratio. In particular the instruction set contains specific instructions handling execution- and context-passing. Communications are guaranteed to respect message ordering and to be error free. In our implementation we use a simplified TCP/IP protocol stack.

The absence of a scheduler and preempted code is a design choice that aims at simplicity. Code is simply executed until it ends or migrates to another site. There is therefore no performance penalty for context switching and no need for a multi-threaded operating system.

An important feature of $_d$SL's execution environment is that the memory size is bounded. Two reasons ensure this property : first, there is no dynamic memory allocation in $_d$SL, so global memory can statically be allocated. Secondly, only a bounded number of processes with a local context coexist during execution. Indeed, each site has one process that handles uninterruptible synchronous code and remotely `LAUNCH`ed methods. Finally, we can remark that since sequences have only one running instance at a time and that stacks are limited to a certain size, resulting in a statically known amount of memory.

In its current state, the $_d$SL compiler/distributer and VM have been implemented in 20k+7k lines of C/C++ code as a proof of concept. The present implementation of the VM fits into 1MB of memory, including the dynamic program loader, debugger and interpreter. Further development, separate compilation, dynamic types, user interface, etc. is taken over by Macq Electronique. The introduction of pointers makes the static distribution process of $_d$SL hard. We are working on a solution that uses extensive pointer analysis. Since $_d$SL is integrated into Macq Electronique's OBviews, a commercial development environment and toolkit for the programming and supervision of PLCs, it benefits from a high number of existing utilities. The localization table as well as the description of types and global variables uses the graphical user interface of OBViews' database subsystem. The $_d$SL Virtual Machine is compatible with the OBViews' supervision subsystem that allows, amongst others, to create graphical representations of the controlled system, stimulated by the state of the controlling $_d$SL program. In [12] we studied a controller, specified by a 200 lines $_d$SL program, for the locks of a canal where various distributed constraints have to be respected (e.g. not to open both gates of the same lock). The resulting VM code was distributed on 3 PLCs. As a proof of concept, we actually built a small scale model of the locks using *Lego Mindstorms*^{Tm} and interfaced the PLCs to the engines and sensors[6].

6 Future work

The main research we are conducting now and will pursue in the future is to enable the formal verification of $_{\mathrm{d}}$SL. A first experiment has been conducted on the canal locks example of [12], where the $_{\mathrm{d}}$SL source code is translated into Promela, and verified for the correctness of safety properties by the Spin model checker. We are working on the automatic translation of $_{\mathrm{d}}$SL to Promela and more generally on an efficient way to verify $_{\mathrm{d}}$SL programs avoiding the state space explosion problem. The indeterminism in $_{\mathrm{d}}$SL is caused by the asynchronous composition and communications but not by the $_{\mathrm{d}}$SL code itself. We hope to be able to use this fact to reduce the state space in addition to the classical methods like abstraction, symmetry and partial order reduction. We are also investigating how lightweight verification can offer a solution between testing and exhaustive verification. In order to achieve this validation, we developed a prototype debugger capable of generating traces. The model checker is used to explore the state space within a certain diameter of those traces.

In the case of exhaustive verification, we need to obtain a closed system. In order to do so, we must build a sufficiently detailed specification of the environment (i.e. the industrial process to control). This problem can be simplified by offering the user a verified library of pre-constructed and parameterizable common environments.

Further topics include efficiency (real-time behavior) and robustness issues.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Inc., Reading, Mass., 1986.
2. P. Aubry. *Mises en oeuvre distribues de programmes synchrones (thèse)*. Phd thesis, IFSIC, Rennes, France, October 1997.
3. E. Balas, G. Cornuéjols, and R. Kannan, eds. Algorithms and min-max theorems for certain multiway cuts. In *Proc. of the 2nd Integer Programming and Combinatorial Optimization Conference*, pages 334–345. Carnegie Mellon University, May 1992.
4. 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.
5. 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.
6. Gerard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
7. H. Brinksma and R. Langerak. Functionality decomposition by compositional correctness preserving transformation. *South African Computer Journal*, 13:2–13, 95.
8. 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.
9. 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.
10. L. G. Demichiel. *Entreprise JavaBeans*^{Tm} *Specification, Version 2.1*. Sun Microsystems, Aug 2002. Proposed Final Draft.
11. Bram DeWachter. Code Distribution in the dsl Environment for the Synthesis of Industrial Process Control. Technical report, U.L.B., 15 January 2003.
12. Bram DeWachter, Thierry Massart, and Cedric Meuter. An experiment on synthesis and verification of an industrial process control in the dsl environment. Proceedings of the 3rd Automated Verification of Critical Systems (AVoCS03), Technical Report DSSE-TR-2003-2, DSSE, Southampton (GB), April 2-3 2003.

[6] Pictures and videos available at http://www.ulb.ac.be/di/ssd/bdewacht/dsl

13. B. Dimitrov and V. Rego. Arachne: A portable threads system supporting migrant threads on heterogeneous network farms. *IEEE Transactions on Parallel and Distributed Systems*, 9(5):459–, 1998.
14. 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.
15. A. Girault. *Sur la Répartition de Programmes Synchrones.* Phd thesis, INPG, Grenoble, France, January 1994.
16. Object Management Group. The common object request broker: Architecture and specification minor revision 2.3.1. OMG TC Document formal/99-10-07, Framingham, MA, USA, 1999.
17. 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.
18. H. Jiang and V. Chaudhary. Compile/run-time support for thread migration. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, Fort Lauderdale, Florida, April 2002.
19. 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.
20. 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.
21. T. Massart. A calculus to define correct transformations of LOTOS specifications. In *Proceedings of the FORTE'91 conference*, pages 281–296, 1992.
22. L. Merrick. Dcom technical overview. Technical report, Microsoft, 1996.
23. R. Milner. On relating synchrony and asynchrony. Technical Report CSR-75-80, Computer Science Dept., Edinburgh Univ., 1981.
24. R. Milner. *Communication and Concurrency.* PHI Series in Computer Science. Prentice Hall, 1989.
25. 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.
26. René Morin. Decompositions of Asynchronous Systems. In *Proc. CONCUR'98, Springer Lect. Notes in Comp. Sci. 1466*, pages 549–565. Springer, 1999.
27. 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.
28. Alin Stefănescu, Javier Esparza, and Anca Muscholl. Syntesis of Distributed Algorithm. In *To appear at 14th international conference on concurrency theory (CONCUR 2003)*, 2003.

A $_{\text{d}}$SL grammar

Note that this grammar is a reduced version of the complete $_{\text{d}}$SL grammar. To keep it compact, some rules may introduce ambiguity.

dsl_program → *decl_list main_program*

main_program → “**PROGRAM**” *var_decl instruction_list* “**END_PROGRAM**”

decl_list → *decl decl_list* | *decl*

decl → *class_decl* | *global_var_decl* | *when_decl* | *method_decl* | *sequence_decl*

class_decl → “**CLASS**” *id var_list* “**END_CLASS**”

global_var_decl → “**GLOBAL_VAR**” *var_list* “**END_VAR**”

when_decl → “**WHEN**” *expression* “**THEN**” *instruction_list* “**END_WHEN**” |
“**WHEN**” “**IN**” *id expression* “**THEN**” *instruction_list* “**END_WHEN**”

method_decl → “**METHOD**” *id* “(” *param_decl* “)” *var_decl instruction_list*
“**END_METHOD**” | “**METHOD**” *id* “::” *id* “(” *param_decl* “)” *var_decl*
instruction_list “**END_METHOD**”

sequence_decl → “**SEQUENCE**” *id* “(” *param_decl* “)” *var_decl instruction_list*
“**END_SEQUENCE**”

var_decl → “**VAR**” *var_list* “**END_VAR**”

param_decl → *id_list* “:” *var_type* | *id_list* “:” *var_type* “,” *param_decl*

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

id_list → *id* | *id* “**,**” *id_list*

var_type → “**INT**” | “**BOOL**”

instruction_list → *instruction instruction_list* | *instruction*

instruction → *if* | *while* | *assign* “;” | *synch_call* “;” | *asynch_call* “;”

if → “**IF**” *expression* “**THEN**” *instruction_list* “**END_IF**” |
“**IF**” *expression* “**THEN**” *instruction_list* “**ELSE**” *instruction_list*
“**END_IF**”

while → “**WHILE**” *expression* “**DO**” *instruction_list* “**END_WHILE**”

assign → *lh_side* “:=” *expression* “;”

lh_side → *id* | *lh_side* “.” *id*

synch_call → *id* “(” *id_list* “)” | *lh_side* “< −” *id* “(” *id_list* “)”

asynch_call → “**LAUNCH**” *synch_call*

expression → *expression binary_op expression* | *unary_op expression* | “(” *expression* “)” |
lh_side | “~” *lh_side* | *constant*

binary_op → “+” | “−” | “*” | “/” | “<” | “>” | “<>” | “<=” | “>=” | “==” | “**AND**” |
“**OR**”

unary_op → “−” | “**NOT**” | “**IS_UNKNOWN**”

constant → *number* | “**TRUE**” | “**FALSE**”

B Maximal distribution

Definition 1 (Sets of variables, expressions and instructions). *Given a $_d$SL program, let $\mathcal{V}$ be the set of all its variables, let $\mathcal{E}$ be the set of all its expressions, and $\mathcal{I}$ the set of all its instructions.*

Definition 2 (Variables accessed in an expression). *Let $Var_{\mathcal{E}} : \mathcal{E} \rightarrow 2^{\mathcal{V}}$ be a function that maps an expression $e \in \mathcal{E}$ to the set of all the non-*tilded *variables appearing in e.*

$$Var_{\mathcal{E}}(e) = \begin{cases} \{v\} & \text{if } e \equiv v \\ Var_{\mathcal{E}}(e_1) \cup Var_{\mathcal{E}}(e_2) & \text{if } e \equiv e_1 \ \textit{binary\_op} \ e_2 \\ Var_{\mathcal{E}}(e_1) & \text{if } e \equiv \text{"("} \ e_1 \ \text{")" or } e \equiv \textit{unary\_op} \ e_1 \\ \emptyset & \text{if } e \equiv \text{"~"} \ e_1 \text{ or } e \equiv \textit{constant} \end{cases}$$

This definition can be extended to sets of expressions $Var_{\mathcal{E}}(E) = \cup_{e \in E} Var_{\mathcal{E}}(e)$.

Definition 3 (Variables accessed in an instruction). *Let $Var_{\mathcal{I}} : \mathcal{I} \rightarrow 2^{\mathcal{V}}$ be a function that maps an instruction $i \in \mathcal{I}$ to the set of all the variables appearing in the immediate constituents of i.*

$$Var_{\mathcal{I}}(i) = \begin{cases} \{v\} \cup Var_{\mathcal{E}}(e) & \text{if } i \equiv \ v \ \text{":="} \ e \ \text{";"} \\ Var_{\mathcal{E}}(e) & \text{if } i \equiv \text{"IF"} \ e \ \text{"THEN"} \ \textit{instruction\_list} \ \text{"END\_IF"} \\ Var_{\mathcal{E}}(e) & \text{if } i \equiv \text{"WHILE"} \ e \ \text{"DO"} \ \textit{instruction\_list} \\ & \qquad\quad \text{"END\_WHILE"} \end{cases}$$

This definition can be extended to sets of instructions $Var_{\mathcal{I}}(I) = \cup_{i \in I} Var_{\mathcal{I}}(i)$.

Definition 4 (Variables accessed in a `WHEN`). *If $\mathcal{W}$ denotes the set of all the `WHEN`s in the $_d$SL program, let $Cond_{\mathcal{W}} : \mathcal{W} \rightarrow \mathcal{E}$ be a function that maps a `WHEN` $w \in \mathcal{W}$ to the triggering expression of w and let $Instr_{\mathcal{W}} : \mathcal{W} \rightarrow 2^{\mathcal{I}}$ be a function that maps a `WHEN` $w \in \mathcal{W}$ to the set of all instructions that can be reached from the body of w, including all instruction in the body of w, all instructions reached through synchronous call from the body of w as well as all instructions reached through every `WHEN` that may be triggered because of an assignment of the body of w, and this recursively. Moreover, let $Var_{\mathcal{W}} : \mathcal{W} \rightarrow 2^{\mathcal{V}}$ denote the set of global variables accessed by a `WHEN`: $Var_{\mathcal{W}}(w) = Var_{\mathcal{E}}(Cond_{\mathcal{W}}(w)) \cup Var_{\mathcal{I}}(Instr_{\mathcal{W}}(w))$.*

Definition 5 (Maximal distribution). *Let $\equiv_v$ be the least equivalence relation on $\mathcal{V}$ such that $\forall v_1, v_2 \in \mathcal{V}$:*

(1) $(\exists i \in \mathcal{I} \cdot \{v_1, v_2\} \subseteq Var_{\mathcal{I}}(i)) \rightarrow (v_1 \equiv_v v_2)$
(2) $(\exists w \in \mathcal{W} \cdot \{v_1, v_2\} \subseteq Var_{\mathcal{W}}(w)) \rightarrow (v_1 \equiv_v v_2)$

The maximal distribution is defined by the set of equivalence classes $\{S_1, ..., S_n\}$ of $\mathcal{V}$ induced by $\equiv_v$. Each S_i defines an execution site.