Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 238 (2009) 3–19
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Modular Distribution and Application to Discrete Controller Synthesis

Gwenaël Delaval[1]

INRIA Grenoble Rhône-Alpes and University of Grenoble, POP ART project team, France

Abstract

This paper shows the application of the automatic distribution of synchronous reactive programs to the specific problem of discrete controller synthesis of complex reactive systems. Discrete controller synthesis is a formal method used to ensure properties on a flexible system which does not a priori verify them. However, this method is efficient only on Boolean programs. More complex embedded systems, comprising complex data types and structures, cannot be addressed without abstraction means. We show how such abstractions can be obtained automatically using a type-directed projection operation. This operation allows then the safe recombination of the result of the synthesis with the original abstracted system, preserving the ensured properties.

Keywords: Discrete controller synthesis, automatic distribution, type systems

1 Introduction

Reactive embedded systems, due to their strong interaction with their environment, are intrinsically critical. The failure of such systems can involve serious human or ecological damages. For these reasons, there is a strong need of formal methods for the design of such systems, in order to ensure formal properties on them.

We address here the problem of ensuring properties by mean of an automatic method of transformation of the original program. Some of these methods cannot be applied on complex and general programs, unless performing a preliminary separation of the program in two parts, one on which the automatic transformation is performed, and one with the other computations. This separation can be fastidious, and the recombination of the two parts raises safety and modularity issues. The contribution of this paper is to use an automatic type-directed distribution method to obtain on one hand an abstraction on which the transformation will be applied, and on the other hand the complementary of this abstraction, whose recombination

[1] Email: Gwenael.Delaval@inria.fr

doi:10.1016/j.entcs.2008.01.003

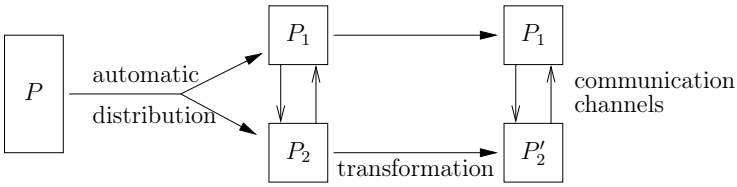


Fig. 1. Automatic distribution for application of a transformation on part of a program.

with the transformed abstraction is safe and preserves the transformation. This work is based on [7], where a type-directed projection operation is defined in order to address the modular distribution of reactive programs, and consists of both an adaptation of this method and an illustration of its application towards a specific distribution problem, namely here separate compilation and analysis of centralized systems.

Figure 1 sums up this method: P is the original program distributed into P_1 and P_2. The transformation is applied on P_2, giving the program P_2', then recombined safely with P_1. We show the application of this method on a specific transformation, the discrete controller synthesis [2], whose goal is to ensure properties on a partial design. This method allows the programming of both parts of a program, for example here the control part and the data part, in the very same language and in an integrated way, thus giving the programmer more control on the semantics of the written programs.

The properties we wish to address here are temporal logic properties like the evolution of values of inputs and outputs of the program during time. An example of such property can be expressed as "the event B will always occur after the event A". These properties are commonly checked by the use of automatic tools (e.g., model-checking tools [10]), on a conservative abstraction of a fully designed system. Another approach lies in discrete controller synthesis (DCS) [17]: given a partially designed system and temporal properties, this method computes automatically a controller, actuating on the partial system, and ensuring the required properties. This method has shown its interest for embedded systems in [2].

However, this method is efficient only on a Boolean state-based abstraction of the system [13]. This abstraction can involve addition of inputs (e.g., Boolean values from comparison of scalar data) and masking of other ones. The use of this method also involves separate compilation of the state-based abstraction of the original system, thus exposing the final system to data or control consistency problems. Moreover, as current DCS algorithms apply only on global programs, the Boolean abstraction needs to be inlined.

We present here an automatic method allowing the generation, from a unique source program, of a Boolean state-based abstraction (named *Boolean fragment* hereafter, on which we will perform the DCS to ensure the required properties), and a program performing the other data computations (named *data fragment*). We show how our method allows the correct automatic recombination of the data fragment with the controlled Boolean fragment, once discrete controller synthesis has been performed on it. This separation of the Boolean and the data fragments also allows

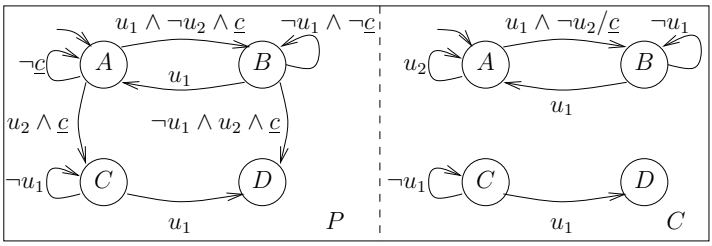


Fig. 2. Example of discrete controller synthesis.

us to only inline the part on which the DCS is applied, thus keeping the original modularity structure of the initial program in the data part, thus allowing, for example, to use higher-order features in it.

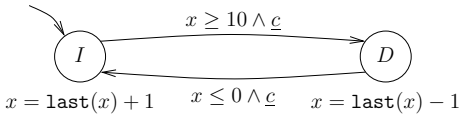
This paper is organized as follows: Section 2 gives an overview of our method. Section 3 shows an application example. Section 4 presents the language used, Section 5 the type-directed projection operation, and Section 6 the application of DCS on the two obtained fragments. Finally, Section 7 discusses our method and presents some prospects.

2 Overview

Discrete controller synthesis operates on Boolean reactive systems, often expressed as automata. Temporal logic properties are then expressed as properties on the set of traces of these automata. Given a partial program P and a property Φ not satisfied *a priori* by P, the discrete controller synthesis computes a controller C, such that the synchronous composition of P and C is a program which satisfies Φ. The partial nature of a program is expressed by additional inputs named *controllable inputs*, whose values will be defined by the computed controller. This set of controllable inputs is denoted I_c, and therefore the discrete controller synthesis operation will be noted $\mathrm{DCS}(P, I_c, \Phi)$. By opposition, actual inputs coming from the program environment are called hereafter *uncontrollable inputs* and denoted I_u. Sigali [13] and Supremica [1] are examples of DCS tools.

Figure 2 shows an example of discrete controller synthesis. Synchronous programs are expressed as Mealy machine. The program P, on the left, has two uncontrollable inputs u_1 and u_2, and a controllable input $\underline{c}$ (underlined, as every controllable input in the following). This program is partially designed: in state A, with the input u_2, two possibilities have been specified as correct by the programmer: staying in state A, or going to state C. Let assume now that, for safety reasons, we want to ensure that the state D will never be reached (expressed as the CTL formula $\Phi = \forall\Box\neg D$). Then, the computation of $\mathrm{DCS}(P, \{c\}, \Phi)$ produces the controller shown on the right of Figure 2, which gives, at occurrences of u_2 in state A, the false value for the controllable input $\underline{c}$. The result of the synchronous product of P and C is a program where, for any sequence of inputs u_1 and u_2, the state D is never reached.

Such methods can only be applied on Boolean state-based systems. Moreover,




```
automaton
| I -> do x = last x + 1
    until (x >= 10) & c then D
| D -> do x = last x - 1
    until (x <= 0) & c then I
end
```

Fig. 3. Mode automata example.

these methods are not very scalable: they cannot handle, without abstraction, more complex systems (e.g., mixing Boolean and numerical values). Two solution exists to alleviate these problems: a Boolean abstraction must either be extracted of the whole system, or it can be designed apart. The first solution can involve arbitrary complex manual work to obtain an abstraction, then the recombination of the controlled abstraction and the original system. This is why the second solution is considered in the framework of the design of embedded systems in computation/control layers [18]. This paradigm allows the design of embedded systems in terms of, on one hand, a computational layer, comprising the implementation of complex functional tasks like control law, signal processing; and on the other hand a control layer, purely reactive, allowing the schedule of the functional tasks.

Thus, these two parts can be separately designed and compiled, and specific analysis tools (model-checking or discrete controller synthesis) can be used for the reactive part. Another interest of this separation approach is to design modular reactive patterns, tested and formally checked once, and reusable for several designs. This approach has shown its interest for domain-specific approchs, like ORCCAD [3], where the Robot-tasks are such modular patterns, or the NEMO language [8].

However, this separate design can be problematic for the global simulation and analysis of the system. Furthermore, it yields a lack of modularity and flexibility of the control interface, especially in the case of use of domain-specific patterns, the approach commonly taken being specific code duplication and inlining. Mode automata [12], as well as further work on synchronous languages [6], has addressed such problem by adding automata as control structures in the dataflow languages Lustre [9] and Lucid Synchrone [5,16].

Figure 3 shows an example of mode automata, with its equivalent specification in Lucid Synchrone. This automaton is composed of two modes or states, one increasing the global variable x (named I), one decreasing it (D). The transition from I to D (resp. D to I) is taken when $x \geq 10$ (resp. $x \leq 0$), and the controllable input $\underline{c}$ is true (this allow the controller to inhibit the transition between the two modes).

These two complementary approaches, global simulation and analysis, and separate compilation, can be both applied on the same system, by means of the application on dataflow languages extended with automata of an automatic distribution method, allowing us to obtain several fragments from a whole program, on which separate analysis and compilation can be applied, before safe recombination. Applying this method for discrete controller synthesis, the goal would be to obtain two fragments, a Boolean abstraction on one hand, and the remaining computations on

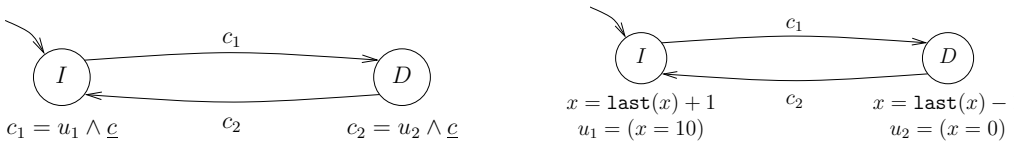


Fig. 4. Two fragments obtained from the program of Figure 3.

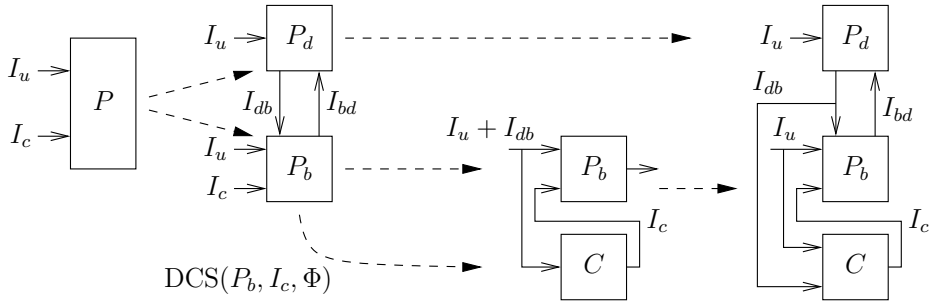


Fig. 5. Distribution for application of discrete controller synthesis.

another. Figure 4 shows the application of this method on the automaton of Figure 3. The Boolean fragment is shown on the left, and the remaining computations are performed by the automaton on the right. For the communication of values from one fragment to the other, u_1 and u_2 are added as outputs of the second automaton, and as uncontrollable inputs on the Boolean fragment. c_1 and c_2 are added conversely to communicate the result of Boolean computations. In the following, we will keep the convention of calling u (resp. c) the channel communications from the data fragment to the Boolean fragment (resp. from the Boolean to the data fragment). The synchronous composition of these two automata is semantically equivalent with the automaton of Figure 3.

Figure 5 sums up our method. Once the two fragments P_b and P_d have been obtained from the original program P, the discrete controller synthesis can be performed on the Boolean fragment P_b, and the composition of this Boolean fragment with the computed controller C can then be safely composed with the other computations obtained by the distribution. I_{db} and I_{bd} are respectively the set of communication channels added from P_d to P_b (so as outputs of P_d and uncontrollable inputs of P_b), and the way back. The synchronous composition of the Boolean fragment with its controller is performed by plugging the inputs I_u and I_{db} as inputs of the controller, and the output of the controller as controllable inputs of the fragment.

3 Application

Figure 6 shows as illustrating example the programming of a pattern named `task`, representing a delayable task. This pattern is a higher-order node: its parameter `f` is the function to be executed by an instantiation of this task when it is active, `req` and `stop` are two Boolean requesting respectively the launch of the task and its end, and `x` is the input of `f`. The task is delayable because an input `ok` can delay it when it is requested. This input will be instantiated with controllable ones. The

outputs of this task are the output `y` of `f`, and a Boolean `act`, true when the task is actually active. The main node then contains two instantiations of this pattern. Programs are here written in the simplified syntax used further in Section 4. The concrete syntax of Lucid Synchrone can be seen in [16].

```
node task(f, req, ok, stop, x) = (y,act) where
  automaton
  | Idle -> do
          act = false
        and y = 0
      until (req & ok) then Active
      until (req & not ok) then Wait
  | Wait -> do
          act = false
        and y = 0
      until ok then Active
  | Active -> do
            act = true
        and y = f(x)
      until stop then Idle
  end;

    y1,act1 = task(f, (x1>0), ok1, (x1<0), x1)
and y2,act2 = task(g, (x2>0), ok2, (x2<0), x2)
```

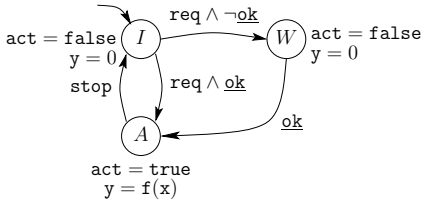


Fig. 6. Example: delayable task pattern.

Performing a type-directed distribution allows for this distribution to be modular. Thus, one node will result in exactly one node on each fragment. In order to do this modular operation, the typing approach allows us to attach to each node the information about where its inputs and outputs are located (i.e., on which fragment they have been computed), and what fragment and what communication channels are involved by this node's computation.

The result of the partition of this program is shown in Figures 7 and 8. On the Boolean fragment (Fig. 7), the node `task` has four additional outputs for each transition. Thus, as this node is instantiated twice in the main node, eight outputs `c1` to `c8` are added. The controller computed on this abstraction can, e.g., ensure that these two tasks will never be active at the same time ($\forall\Box\neg(\texttt{act1} \wedge \texttt{act2})$).

The abstraction with other computations is given in Figure 8. The outputs added in the Boolean abstraction are here added as inputs, and new outputs are added, holding the values of comparisons (`u1` to `u4`).

Both fragments comprise an automaton, whose states and transitions match with the original one. The targeted behaviour is a Boolean fragment computing transitions, and a data fragment receiving scheduling instructions under the form of triggered transitions between computing states. Unused values (`f` and `x` on the Boolean fragment, `req`, `ok` and `stop` on the data one) are replaced by the special value `()`.

```
node task(f, req, ok, stop, x) = ((),act,c1,c2,c3,c4) where
  automaton
  | Idle -> do
          act = false
       and c1 = (req & ok)
       and c2 = (req & not ok)
      until c1 then Active
      until c2 then Wait
  | Wait -> do
          act = false
       and c3 = ok
      until c3 then Active
  | Active -> do
            act = true
        and c4 = stop
      until c4 then Idle
  end;

    y1,act1,c1,c2,c3,c4 = task((), u1, ok1, u2, ())
and y2,act2,c5,c6,c7,c8 = task((), u3, ok2, u4, ())
```

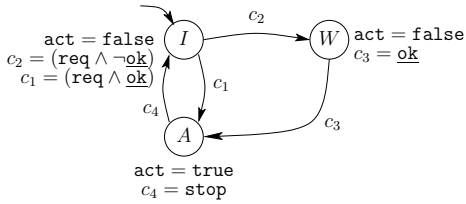


Fig. 7. Boolean fragment of Figure 6.

```
node task(f, req, ok, stop, x, c1, c2, c3, c4)
        = (y,()) where
  automaton
  | Idle -> do y = 0
      until c1 then Active
      until c2 then Wait
  | Wait -> do y = 0
      until c3 then Active
  | Active -> do y = f(x)
      until c4 then Idle
  end;

    u1 = (x1>0)
and u2 = (x1<0)
and y1,act1 = task(f, (), (), (), x1, c1, c2, c3, c4)
and u3 = (x2>0)
and u4 = (x2<0)
and y2,act2 = task(g, (), (), (), x2, c5, c6, c7, c8)
```

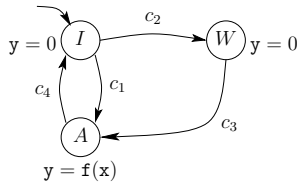


Fig. 8. Data fragment of Figure 6.

4 The language

The language we consider is a dataflow language, extended with automata. It follows the following syntax:

$$
\begin{aligned}
P &::= d_1\,;\ldots;d_n\,;D \\
d &::= \texttt{node}\ f(p) = e\ \texttt{where}\ D \\
p &::= p, p \mid x \\
D &::= \epsilon \mid p = e \mid p = x(e) \mid D\ \texttt{and}\ D \mid \texttt{automaton}\ S \to h \ldots S \to h\ \texttt{end} \\
h &::= \texttt{do}\ D\ u \\
u &::= \texttt{until}\ e\ \texttt{then}\ S\ u \mid \epsilon \\
e &::= i \mid x \mid (e, e) \mid \mathrm{op}(e, e) \mid e\ \texttt{fby}\ e
\end{aligned}
$$

A program P is a sequence of node definitions, followed by a set of equations (the "main" of the program). A node definition d takes as input a pattern p, comprising

a sequence of uncontrollable inputs (x). Its output is an expression e, evaluated within the scope of a set of equations D.

Definitions D are either an empty definition ϵ, single equations defining patterns of variables ($p = e$), definitions naming the result of an application ($p = x(e)$), parallel declarations (D `and` D), or an automaton, composed of a sequence of states S defined by one handler h (`automaton` $S \to h \ldots S \to h$ `end`). An automaton handler is a definition followed by a sequence of transitions.

An expression e may be an immediate value (i), a variable (x), a pair construction (e, e), a binary combinatory operation (op(e, e), where op can be $+$, $-$,$\ldots$), an initialized delay (e `fby` e). The pair construction is left-associative, and thus we denote by $e_1, \ldots, e_n$ the expression $(\ldots(e_1, e_2), e_3), \ldots), e_n)$.

5 Type-directed projection

The type system is adapted from [7]. It is extended with automata, and simplified to answer the specific problem of separate compilation of a program in two parts (i.e., two locations). The target architecture here is then composed of two "locations", named A_b and A_d (A_b will be the Boolean fragment and A_d the data fragment), which are connected to each other in both directions.

Our type system associates to each expression and definition a *spatial type*, which expresses on which fragment (Boolean or data) this expression or definition is located or defined. For example, a value of spatial type `bool at` A_d is a Boolean value located (and then, usable) on the data fragment. The spatial type of a node taking as input a Boolean, whose output is a Boolean, and which is only computable on the Boolean fragment without internal communication, will be of spatial type `bool at` A_b $\multimap\!\langle\{A_b\}/\emptyset\rangle\!\rightarrowtail$ `bool at` A_b. It is a type and effect system [19], as with its spatial type, we associate to each expression and definition the set of fragments, and the channels involved in its computation.

As a more complete example, the type of the node `task` of Figure 6 will carry the fact that it takes as input a node whose input, output, and computation are on A_d (the data fragment), three inputs on A_b (the Boolean fragment) and one on A_d, that its output is a pair whose first component is on A_d and its second one on A_b, and that the computation of `task` involves the two fragments A_b and A_d and four communication channels from A_b to A_d:

$$\forall\alpha.\Big((\alpha \texttt{ at } A_d \multimap\!\langle\{A_d\}/\emptyset\rangle\!\rightarrowtail \texttt{int at } A_d) \times \texttt{bool at } A_b \times \texttt{bool at } A_b \times \texttt{bool at } A_b \times \alpha \texttt{ at } A_d\Big) \multimap\!\langle\{A_b, A_d\}/T\rangle\!\rightarrowtail (\texttt{int at } A_d \times \texttt{bool at } A_b)$$
$$\text{where } T = [A_b \overset{c_1}{\mapsto} A_d, A_b \overset{c_2}{\mapsto} A_d, A_b \overset{c_3}{\mapsto} A_d, A_b \overset{c_4}{\mapsto} A_d]$$

The syntax of spatial type expressions is:

$$\begin{array}{ll} \sigma ::= \forall \alpha_1, \ldots, \alpha_n . \forall \delta_1, \ldots, \delta_p . t & b ::= \texttt{int} \mid \texttt{bool} \mid \ldots \\ t ::= b \mid \alpha \mid t \texttt{ at } s \mid t \multimap\!\langle \ell / T \rangle\!\rightarrow t \mid t \times t & s ::= A_b \mid A_d \mid \delta \\ T ::= [A_1 \overset{c_1}{\mapsto} A'_1, \ldots, A_n \overset{c_n}{\mapsto} A'_n] \text{ if } \forall i \neq j, c_i \neq c_j & \ell ::= \{s_1, \ldots, s_n\} \\ H ::= [x_1 : \sigma_1, \ldots, x_n : \sigma_n] \text{ if } \forall i \neq j, x_i \neq x_j & \end{array}$$

H denotes spatial typing environments. We distinguish spatial type schemes (σ), which can be quantified, from simple spatial types (t). A simple spatial type can be either a stream base type (b, like integers, Booleans, ...), a type variable (α), a located type (t `at` s), a node type ($t \multimap\!\langle \ell / T \rangle\!\rightarrow t$), or a pair type ($t \times t$). A location is either one of the two constant location A_b (for the Boolean fragment) and A_d (for the data fragment), or a location variable δ. ℓ denotes sets of locations (hence limited here to subsets of $\{A_b, A_d\}$, or singleton $\{\delta\}$).

A value of spatial type t `at` s is a value located on s. A value of spatial type $t_1 \multimap\!\langle \ell / T \rangle\!\rightarrow t_2$ is a node whose input is of spatial type t_1, whose output is of spatial type t_2, and whose computation involves the set of locations ℓ.

So as to allow, e.g., the instantiation of a type scheme of the form $\forall \alpha . \alpha$ `at` s by the type `int at` $s \multimap\!\langle \{s\} / \emptyset \rangle\!\rightarrow$ `int at` s, we extend the equality relation between types. Thus, the following equalities stand:

$$\begin{array}{c} (t_1 \times t_2) \texttt{ at } s = (t_1 \texttt{ at } s) \times (t_2 \texttt{ at } s) \\ (t_1 \multimap\!\langle \{s\} / \emptyset \rangle\!\rightarrow t_2) \texttt{ at } s = (t_1 \texttt{ at } s) \multimap\!\langle \{s\} / \emptyset \rangle\!\rightarrow (t_2 \texttt{ at } s) \\ t \texttt{ at } s \texttt{ at } s = t \texttt{ at } s \end{array}$$

$$\begin{array}{cc} t_1 = t'_1 \quad t_2 = t'_2 & t_1 = t'_1 \quad t_2 = t'_2 \\ (t_1 \times t_2) = (t'_1 \times t'_2) & t_1 \multimap\!\langle \ell / T \rangle\!\rightarrow t_2 = t'_1 \multimap\!\langle \ell / T \rangle\!\rightarrow t'_2 \end{array}$$

The notion of *well-formed* types is introduced, because one cannot associate a meaning to any type: e.g., the spacial type `int at` A_b `at` A_d cannot appear in the type system. A spatial type t is *well-formed* iff $\forall t', s_1, s_2, t = t'$ `at` s_1 `at` $s_2 \Rightarrow s_1 = s_2$. Every type in the type system is assumed to be well-formed, and typing algorithms maintain this invariant during typing.

The instantiation mechanism is defined as follows:

$$t[t_1/\alpha_1, \ldots, t_n/\alpha_n, s_1/\delta_1, \ldots, s_p/\delta_p] \leq \forall \alpha_1 \ldots \alpha_n . \forall \delta_1 \ldots \delta_p . t$$

The generalization mechanism is restrained for the projection. We allow type schemes to comprise at most one location variable. In that case, the type scheme is of the form $\forall \alpha_1 \ldots \alpha_n . \forall \delta . t$ `at` δ, and the projection consists in placing the value of this type on the two fragments.

We note respectively $\mathrm{FLV}(t)$ and $\mathrm{FTV}(t)$ the set of free location variables and free type variables of the type t. FLV and FTV are straightforwardly extended to typing environments. The generalization of t in typing environment H is noted

$\text{gen}_H(t)$, defined as:

$$\text{gen}_H(t) = \forall\alpha_1,\ldots,\alpha_n.\forall\delta_1,\ldots,\delta_p.t \text{ where } \begin{array}{l}\{\alpha_1,\ldots,\alpha_n\} = \text{FTV}(t) - \text{FTV}(H)\\ \{\delta_1,\ldots,\delta_p\} = \text{FLV}(t) - \text{FLV}(H)\\ p \leq 1\end{array}$$

The function locations$(\cdot)$ gives the set of locations involved in the spatial type given as argument. It is defined as:

$$\begin{array}{ll}\text{locations}(t_1 \times t_2) & = \text{locations}(t_1) \cup \text{locations}(t_2)\\ \text{locations}(t_1 \multimap\!\langle \ell/T \rangle\!\rightarrow t_2) & = \ell\\ \text{locations}(t \texttt{ at } s) & = \{s\}\end{array}$$

This type system will also infer communication channels needed between the two fragments. A channel will be represented by added common variables between two fragments, i.e., added inputs on one side corresponding to added outputs on the other. Channels are named so as to be able to match these added variables in the two fragments. Formally, a channel is a location pair associated with a name, noted $A_1 \overset{c}{\mapsto} A_2$: c is the name of the channel, A_1 its source location, and A_2 its destination location. c is the common variable representing a communication between the two fragments. The set of channel names is totally ordered, so as to keep consistency of inputs and outputs added, from the node definition to node instances. T denotes channels sequences, in which channel names are disjoints. (T, T') denotes the concatenation of two channel sequences. $\text{dom}(T)$ is the set of channel names of T. For the sake of multiple instanciation of nodes, it is necessary to be able to rename channels, so as to keep these names disjoints in inferred channel sequences. Given two channel sets T and T', we note $T' \cong T$ the fact that T' is equal to T modulo a renaming of its channels:

$$T' \cong T \Leftrightarrow T' = T[c'_1/c_1,\ldots,c'_n/c_n] \qquad \begin{array}{l}\text{where } \text{dom}(T) = \{c_1,\ldots,c_n\}\\ \text{and } \text{dom}(T') = \{c'_1,\ldots,c'_n\}\end{array}$$

Then, from a channel sequence T, we denote by $T \uparrow s$ (resp., $T \downarrow s$) the channel sequence extracted from T and which source (resp., destination) is s.

$$\begin{array}{c}\emptyset \uparrow s = \emptyset\\ ([s \overset{c}{\mapsto} s_2], T) \uparrow s = [s \overset{c}{\mapsto} s_2], (T \uparrow s)\\ ([s_1 \overset{c}{\mapsto} s_2], T) \uparrow s = (T \uparrow s) \text{ iff } s_1 \neq s\end{array} \qquad \begin{array}{c}\emptyset \downarrow s = \emptyset\\ ([s_1 \overset{c}{\mapsto} s], T) \downarrow s = [s_1 \overset{c}{\mapsto} s], (T \downarrow s)\\ ([s_1 \overset{c}{\mapsto} s_2], T) \downarrow s = (T \downarrow s) \text{ iff } s_2 \neq s\end{array}$$

Finally, a program is typed and projected with the initial environment $H_0 = H_{\text{op}}, H_u, H_c$, where H_{op} contains the types of the initially defined operations (Boolean operations being defined to be computed on A_b, and other on A_d), and H_u and H_c

respectively the types of the uncontrollable and controllable inputs.

$$H_{\text{op}} = \begin{bmatrix} \cdot\ \texttt{fby}\ \cdot : \forall\alpha.\forall\delta.\alpha\ \texttt{at}\ \delta \times \alpha\ \texttt{at}\ \delta \multimap\!\langle\{\delta\}/\emptyset\rangle\!\rightarrow \alpha\ \texttt{at}\ \delta, \\ \texttt{fst}\ \cdot : \forall\alpha,\beta.\forall\delta_1,\delta_2.\alpha\ \texttt{at}\ \delta_1 \times \beta\ \texttt{at}\ \delta_2 \multimap\!\langle\{\delta_1,\delta_2\}/\emptyset\rangle\!\rightarrow \alpha\ \texttt{at}\ \delta_1, \\ \texttt{snd}\ \cdot : \forall\alpha,\beta.\forall\delta_1,\delta_2.\alpha\ \texttt{at}\ \delta_1 \times \beta\ \texttt{at}\ \delta_2 \multimap\!\langle\{\delta_1,\delta_2\}/\emptyset\rangle\!\rightarrow \beta\ \texttt{at}\ \delta_2, \\ (\texttt{and}) : \texttt{bool at}\ A_b \times \texttt{bool at}\ A_b \multimap\!\langle\{A_b\}/\emptyset\rangle\!\rightarrow \texttt{bool at}\ A_b, \ldots \\ (+) : \texttt{int at}\ A_d \times \texttt{int at}\ A_d \multimap\!\langle\{A_d\}/\emptyset\rangle\!\rightarrow \texttt{int at}\ A_d, \ldots \end{bmatrix}$$

$$H_u = [x_{u1} : t_1\ \texttt{at}\ s_n, \ldots, x_{un} : t_n\ \texttt{at}\ s_n]$$

$$H_c = [x_{c1} : \texttt{bool at}\ A_b, \ldots, x_{cp} : \texttt{bool at}\ A_b]$$

We note I_u and I_c respectively the sets of uncontrollable and controllable inputs: $I_u = \text{dom}(H_u)$ and $I_c = \text{dom}(H_c)$.

The type system defines a judgment $H \vdash D : H'/\ell/T$, meaning that in teh typing environment H, the definition D yields the typing environment H', and its computation involves the set of locations ℓ, and the communication channels T. This type system defines a type directed projection operation, performed on a typed program. For the sake of brevity, we will detail here only the rules of this projection operation. The rules defining the type system can be found in appendix.

The projection of a definition D on a location s is noted $H \vdash D : H'/\ell/T \overset{s}{\Longrightarrow} D'$, meaning that in the typing environment H, the definition D defines the environment H', and its projection on s results in the new definition D'. ℓ is the set of locations involved, and T is the set of communication channels (i.e., added inputs and outputs) needed for the execution of D. The set of locations involved is used for the suppression of unused definitions and expressions.

This projection operation is straightforwardly extended to programs. Projection of expressions and transitions are noted respectively $H \vdash e : t/\ell/T \overset{s}{\Longrightarrow} e'/D$ and $H \vdash u : \ell/T \overset{s}{\Longrightarrow} u'/D$, where D is in both cases an additional definition of outputs for communications occurring within e or u.

The type system is given in Figures 9 and 10. An immediate value is located on any projected program (rule Imm). A projected variable results in itself iff it is located on the computed fragment (rule Inst).

The rules Comm-From and Comm-To define a subtyping mechanism for inserting communications between the two fragments. An expression e of type t `at` s, which is a value entirely located on s, can be communicated to the other fragment s', and thus can be considered as being of type t `at` s'. The rule Comm-From defines a new communication channel as the value of e, and suppresses this expression in the projection. The rule Comm-To replaces this expression by this new communication channel. For example, this rule is applied, on the program of Figure 6, after the computation of `(x1>0)` on the data fragment. The channel added by this application is $A_d \overset{u_1}{\mapsto} A_b$.

The projection of a pair is the projection of its compounds, new definitions of these compounds being put in parallel (rule Pair). The projection of an equation involves computing the projection of the expression, and putting in parallel with the

projected equation the set of definitions of the communications channels (rule DEF).

The rules SUPPR-EXPR and SUPPR-DEF allow the suppression of any expression on the fragment where no part of this expression is involved.

Projection of a node (rule NODE) involves projecting its compounds, and then adding the communication channels used in these compounds as inputs and outputs of the resulting node, so as to allow multiple instantiations of this node : for example, c_1 to c_4 are added as outputs of the node `task` on the Boolean fragment (see Figure 7). The rules APP-KEEP and APP-SUPPR state that an application is kept iff the fragment computed appears in the set of fragments involved in the computation of the applied node. If it is kept, then the set of communication channels involved within this node are added as inputs and outputs of the application.

Figure 10 shows the rule for the projection of automata. The rules TRANS-D and TRANS-B state that an expression triggering a transition, once computed, must be located on the Boolean fragment. A communication channel is then added, from the Boolean fragment to the data one, holding the value of this expression (c_1 to c_4 on Figure 7). The projected transition holds then the expression composed only of the value of this channel. This channel is computed in the state where the transition can be triggered (rule HANDLER). Then, the projection of an automaton involves the projection of every handler (rule AUTOMATON).

Given two programs $P = d_1\,;\ldots;d_n\,;D$ and $P' = d'_1\,;\ldots;d'_p\,;D'$, we note $P\|P'$ the synchronous composition of P and P', defined as:

$$(d_1\,;\ldots;d_n\,;D)\|(d'_1\,;\ldots;d'_p\,;D') = d_1\,;\ldots;d_n\,;d'_1\,;\ldots;d'_p\,;(D \texttt{ and } D')$$

The semantics of a program P is given by a relation noted $S^i \vdash P : S^o$, meaning that, given the sequence of inputs $S^i = R^i_1.R^i_2.\ldots$, the execution of the program P results in the sequence of outputs $S^o = R^o_1.R^o_2.\ldots$, where R denotes a reaction environment, mapping names to instantaneous values. The semantics is taken from [6] and not rewritten here.

The following property is a refinement of the result stated in [7]. It states that the composition of the two fragments obtained by projection has the same semantics as the original program.

Property 5.1 (Semantical equivalence) *For all S^i, S^o, P, H, H', ℓ and T such that $S^i \vdash P : S^o$, $H \vdash P : H'/\ell/T \overset{A}{\Longrightarrow} P_A$ and $H \vdash P : H'/\ell/T \overset{B}{\Longrightarrow} P_B$, then $S^i \vdash P_A\|P_B : S^o$.*

6 Application of Discrete Controller Synthesis

The formal properties to be ensured by the final program will be expressed as properties on the sequence of outputs S_o. Given a CTL property Φ, we note $P \models \Phi$ the fact that the program P satisfies Φ. We only consider *safety properties*, i.e., properties of the form $\Phi = \forall\Box\varphi$, φ being a predicate on reaction environments. For instance, the property of exclusivity of the two tasks in the program of Section 3 can be expressed as $\forall\Box\varphi$, with $\varphi(R) = (R(\texttt{act1}) = \texttt{false} \vee R(\texttt{act2}) = \texttt{false})$.

$$(\textsc{Imm})\ H \vdash i : b\ \texttt{at}\ s/\{s\}/\emptyset \overset{s}{\Longrightarrow} i/\epsilon \qquad (\textsc{Inst})\ \frac{t \leq (H(x)) \qquad s \in \text{locations}(t)}{H \vdash x : t/\text{locations}(t)/\emptyset \overset{s}{\Longrightarrow} x/\epsilon}$$

$$(\textsc{Comm-From})\ \frac{H \vdash e : t\ \texttt{at}\ s/\ell/T \overset{s}{\Longrightarrow} e'/D}{H \vdash e : t\ \texttt{at}\ s'/\ell \cup \{s'\}/T, [s \overset{c}{\mapsto} s'] \overset{s}{\Longrightarrow} ()/D\ \texttt{and}\ c_n = e'}$$

$$(\textsc{Comm-To})\ \frac{H \vdash e : t\ \texttt{at}\ s/\ell/T \overset{s'}{\Longrightarrow} e'/D}{H \vdash e : t\ \texttt{at}\ s'/\ell \cup \{s'\}/T, [s \overset{c}{\mapsto} s'] \overset{s'}{\Longrightarrow} c_n/D}$$

$$(\textsc{Pair})\ \frac{H \vdash e_1 : t_1/\ell_1/T_1 \overset{s}{\Longrightarrow} e_1'/D_1 \qquad H \vdash e_2 : t_2/\ell_2/T_2 \overset{s}{\Longrightarrow} e_2'/D_2}{H \vdash e_1, e_2 : t_1 \times t_2/\ell_1 \cup \ell_2/T_1, T_2 \overset{s}{\Longrightarrow} e_1', e_2'/D_1\ \texttt{and}\ D_2}$$

$$(\textsc{Suppr-Expr})\ \frac{s \notin \ell}{H \vdash e : t/\ell/T \overset{s}{\Longrightarrow} ()/\epsilon} \qquad (\textsc{Suppr-Def})\ \frac{s \notin \ell}{H \vdash D : H'/\ell/T \overset{s}{\Longrightarrow} \epsilon}$$

$$(\textsc{Def})\ \frac{H \vdash e : t_1 \times \ldots \times t_n/\ell/T \overset{s}{\Longrightarrow} e'/D}{H \vdash (x_1, \ldots, x_n) = e : [t_1/x_1, \ldots, t_n/x_n]/\ell/T \overset{s}{\Longrightarrow} x = e'\ \texttt{and}\ D}$$

$$(\textsc{And})\ \frac{H \vdash D_1 : H_1/\ell_1/T_1 \overset{s}{\Longrightarrow} D_1' \qquad H \vdash D_2 : H_2/\ell_2/T_2 \overset{s}{\Longrightarrow} D_2'}{H \vdash D_1\ \texttt{and}\ D_2 : H_1, H_2/\ell_1 \cup \ell_2/T_1, T_2 \overset{s}{\Longrightarrow} D_1'\ \texttt{and}\ D_2'}$$

$$(\textsc{Node})\ \frac{\begin{array}{c} H, x_i : t_i, H_1 \vdash D : H_1/\ell_1/T_1 \overset{s}{\Longrightarrow} D' \qquad H, x_i : t_i, H_1 \vdash e : t/\ell_2/T_2 \overset{s}{\Longrightarrow} e'/D_e \\ (T_1, T_2) \uparrow s = [s \overset{c_1}{\mapsto} s_1, \ldots, s \overset{c_n}{\mapsto} s_n] \qquad (T_1, T_2) \downarrow s = [s_1' \overset{c_1'}{\mapsto} s, \ldots, s_p' \overset{c_p'}{\mapsto} s] \end{array}}{\begin{array}{c} H \vdash \texttt{node}\ f(x_1, \ldots, x_n) = e\ \texttt{with}\ D : [\text{gen}_H((t_1 \times \ldots \times t_n) \multimap\langle \ell_1/T_1, T_2\rangle\rightarrowtail t)/f]/\ell_1 \cup \ell_2/\emptyset \\ \overset{s}{\Longrightarrow} \texttt{node}\ f(x_1, \ldots, x_n, c_1', \ldots, c_p') = (e', c_1, \ldots, c_n)\ \texttt{with}\ D'\ \texttt{and}\ D_e \end{array}}$$

$$(\textsc{App-Keep})\ \frac{\begin{array}{c} H \vdash f : t \multimap\langle \ell_1/T_1\rangle\rightarrowtail (t_1 \times \ldots \times t_n)/\ell_2/T_2 \overset{s}{\Longrightarrow} f'/D_1 \qquad H \vdash e : t/\ell_3/T_3 \overset{s}{\Longrightarrow} e'/D_2 \\ s \in \ell_1 \qquad T_1' \cong T_1 \qquad T_1' \uparrow s = [s \overset{c_1}{\mapsto} s_1, \ldots, s \overset{c_n}{\mapsto} s_n] \qquad T_1' \downarrow s = [s_1' \overset{c_1'}{\mapsto} s, \ldots, s_p' \overset{c_p'}{\mapsto} s] \end{array}}{\begin{array}{c} H \vdash x = f(e) : [t_1/x_1, \ldots, t_n/x_n]/\ell_1 \cup \ell_2 \cup \ell_3/T_1', T_2, T_3 \\ \overset{s}{\Longrightarrow} (x_1, \ldots, x_n, c_1, \ldots, c_n) = f'(e', c_1', \ldots, c_p')\ \texttt{and}\ D_1\ \texttt{and}\ D_2 \end{array}}$$

$$(\textsc{App-Suppr})\ \frac{\begin{array}{c} H \vdash f : t \multimap\langle \ell_1/T_1\rangle\rightarrowtail (t_1 \times \ldots \times t_n)/\ell_2/T_2 \overset{s}{\Longrightarrow} f'/D_1 \\ H \vdash e : t/\ell_3/T_3 \overset{s}{\Longrightarrow} e'/D_2 \qquad s \notin \ell_1 \qquad T_1' \cong T_1 \end{array}}{H \vdash p = f(e) : [t_2/x]/\ell_1 \cup \ell_2 \cup \ell_3/T_1', T_2, T_3 \overset{s}{\Longrightarrow} D_1\ \texttt{and}\ D_2}$$

Fig. 9. Rules for the projection operation.

Definition 6.1 (Safety property satisfaction)

- A sequence $S = R_1.R_2.\ldots$ satisfies a property $\Box\varphi$ (noted $S \models \Box\varphi$) iff $\forall i, \varphi(R_i)$.
- A program P satisfies a property $\forall\Box\varphi$ (noted $P \models \forall\Box\varphi$) iff for all S^i, S^o such that $S^i \vdash P : S^o$, $S^o \models \Box\varphi$.

The following property states that safety properties are preserved by synchronous composition: given a synchronous program P satisfying a safety property Φ, any synchronous composition $P \| P'$ satisfies Φ. This result has been stated in [10].

$$(\textsc{Trans-D})\ \frac{H \vdash e : t \texttt{ at } A_b/\ell_1/T_1 \overset{A_d}{\Longrightarrow} e'/D_1 \qquad H \vdash u : \ell_2/T_2 \overset{A_d}{\Longrightarrow} u'/D_2}{H \vdash \texttt{ until } e \texttt{ then } S\ u : \ell_1 \cup \ell_2/T_1, T_2, [A_b \overset{c}{\hookrightarrow} A_d] \overset{A_d}{\Longrightarrow} \texttt{ until } c \texttt{ then } S\ u'/D_1 \texttt{ and } D_2}$$

$$(\textsc{Trans-B})\ \frac{H \vdash e : t \texttt{ at } A_b/\ell_1/T_1 \overset{A_b}{\Longrightarrow} e'/D_1 \qquad H \vdash u : \ell_2/T_2 \overset{A_b}{\Longrightarrow} u'/D_2}{H \vdash \texttt{ until } e \texttt{ then } S\ u : \ell_1 \cup \ell_2/T_1, T_2, [A_b \overset{c}{\hookrightarrow} A_d] \overset{A_b}{\Longrightarrow} \texttt{ until } c_n \texttt{ then } S\ u'/c = e \texttt{ and } D_1 \texttt{ and } D_2}$$

$$(\textsc{Handler})\ \frac{H \vdash D : H'/\ell_1/T_1 \overset{s}{\Longrightarrow} D_1 \qquad H \vdash u : \ell_2/T_2 \overset{s}{\Longrightarrow} u'/D_2}{H \vdash D\ u : H'/\ell_1 \cup \ell_2/T_1, T_2 \overset{s}{\Longrightarrow} D_1 \texttt{ and } D_2\ u}$$

$$(\textsc{Automaton})\ \frac{\begin{array}{c}\forall i \in \{1, \ldots, n\}, H \vdash h_i : H_i/\ell_i/T_i \overset{s}{\Longrightarrow} h_i' \\ \ell = \ell_1 \cup \ldots \cup \ell_n \qquad T = T_1, \ldots, T_n \qquad H' = \text{merge}(H_1, \ldots, H_n)\end{array}}{H \vdash \texttt{automaton } S_1 \to h_1 \ldots S_n \to h_n \texttt{ end} : H'/\ell/T \overset{s}{\Longrightarrow} \texttt{automaton } S_1 \to h_1' \ldots S_n \to h_n' \texttt{ end}}$$

Fig. 10. Rules for the projection operation (Automata).

Property 6.2 (Conservation of safety properties)

$$\forall P, \Phi, P', P \models \Phi \Rightarrow (P \| P') \models \Phi.$$

Definition 6.3 (Discrete controller synthesis) DCS is a partial function such that, given a program P, a set of controllable inputs I_c of P, and a safety property Φ, then if $\text{DCS}(P, I_c, \Phi)$ is defined, then $(P \| \text{DCS}(P, I_c, \Phi)) \models \Phi$.

The following result is the formalization of the whole method. It states how, from a program P, a set of controllable inputs I_c, and a safety property Φ, a semantically compatible program P' can be obtained, satisfying Φ. P and P' are not strictly semantically equivalent, as I_c are inputs of P but local variables (defined by the controller) of P'. We will also define a similarity relation $\leq_I$ between programs as follows (where $(R_1.R_2.\ldots) \setminus I = (R_1 \setminus I).(R_2 \setminus I).\ldots$, and $R \setminus I = R'$ iff $\text{dom}(R') = \text{dom}(R) \setminus I$ and $\forall x \in \text{dom}(R'), R'(x) = R(x)$):

$$P' \leq_I P \text{ iff } \forall S_1^i, S^o, S_1^i \vdash P' : S^o \Rightarrow \exists S^i, S^i \setminus I = S_1^i \wedge S^i \vdash P : S^o$$

Theorem 6.4 *For all $P, I = I_u \uplus I_c, \Phi$, let P_b, P_d, H, T, ℓ such that $H_0 \vdash P : H/\ell/T \overset{A_b}{\Longrightarrow} P_b$ and $H_0 \vdash P : H/\ell/T \overset{A_d}{\Longrightarrow} P_d$, then if $C = \text{DCS}(P_b, I_c, \Phi)$ is defined, then let $P' = (P_d \| P_b \| C)$, $P' \leq_{I_c} P$ and $P' \models \Phi$.*

Proof. From property 5.1, $P_b \| P_d$ is semantically equivalent with P. Then, for any program C whose output domain is I, $(P_b \| P_d \| C) \leq_I P$.

Then, $(P_b \| P_d \| \text{DCS}(P_b, I_c, \Phi)) \leq_I P$.

By definition of DCS, $(P_b \| \text{DCS}(P_b, I_c, \Phi)) \models \Phi$. Then from property 6.2, for all P, $(P \| P_b \| \text{DCS}(P_b, I_c, \Phi)) \models \Phi$. Then, $(P_d \| P_b \| \text{DCS}(P_b, I_c, \Phi)) \models \Phi$. □

7 Discussion

We have shown an adaption of an automatic distribution method to allow the separate compilation and transformation of a program. This adaptation has been illustrated on the specific program transformation of discrete controller synthesis. From an initial program, we have shown how to extract the Boolean and the data fragments, how to apply DCS on the Boolean fragment to ensure a safety property, and finally how to recombine by synchronous product the controlled Boolean fragment and the data fragment in such a way that the semantics of the resulting program is ensured to be a simulation refinement of the initial one.

Compared with [7], this article, besides being an application of this method, shows how it can be adapted towards the distribution of programs on an heterogeneous architecture (i.e., whose computing resources does not all provide the same operations), instead of homogeneous ones, as targetted in this previous work. This adaptation consists, for the declaration of the architecture, to attach specific operations to each declared location.

More generally, this example illustrates how a tool allowing co-design of different parts of programs, such as Orccad [3], Ptolemy [4], can integrate two approaches, the design by a unified language or paradigm allowing more control of the program semantics, and the separate compilation, program transformation or analysis. From this point of view, this work can be related to the different approaches of unification of different paradigms in one language (modes and dataflow in Mode automata [11], and its generalization in Lucid Synchrone [6]). The purpose of this work is to show how such language integrations does not stand as an obstacle for the further application of heterogeneous compilation paradigms. Such comparable distribution method has been applied for the design of multi-tier systems [15]: the functions are then annotated with the location where it is executed. We show here how these annotations can be integrated in a language, and how a type system can be used to infer the location of every value or expression.

The prospects of this work lies mainly in considering modular program transformations and analysis. From the point of view of the specific problem addressed here, controller synthesis for hierarchical systems has been studied in [14], and the interaction of this method with modular compilation should be studied. The interest of this method for other mixed models shall also be evaluated.

References

[1] K. Akesson, Martin Fabian, Hugo Flordal, and Arash Vahidi. Supremica — a tool for verification and synthesis of discrete event supervisors. In *Proc. of the 11th Mediterranean Conference on Control and Automation*, Rhodos, Greece, 2003.

[2] K. Altisen, A. Clodic, F. Maraninchi, and E. Rutten. Using controller synthesis to build property-enforcing layers. In *European Symposium on Programming (ESOP)*, April 2003.

[3] J.-J. Borrelly, E. Coste-Manière, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro. The Orccad architecture. *Int. J. of Robotics Research*, 17(4), 1998.

[4] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *Int. Journal in Computer Simulation*, 4(2):155–182, 1994.

[5] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 226–238, New York, NY, USA, 1996. ACM Press.

[6] J.-L. Colaço, B. Pagano, and M. Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005.

[7] G. Delaval, A. Girault, and M. Pouzet. A type system for the automatic distribution of higher-order synchronous dataflow programs. Rapport de recherche INRIA n°6378, November 2007.

[8] G. Delaval and É. Rutten. A domain-specific language for multitask systems, applying discrete controller synthesis. *EURASIP Journal on Embedded Systems*, 2007:Article ID 84192, 17 pages, 2007.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language Lustre. *Proc. of the IEEE*, 79(9):1305–1320, September 1991.

[10] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[11] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming (ESOP)*, Lisbon (Portugal), March 1998. Springer verlag.

[12] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, (46):219–254, 2003.

[13] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4), October 2000.

[14] H. Marchand and B. Gaudin. Supervisory control problems of hierarchical finite state machines. In *41th IEEE Conference on Decision and Control*, Las Vegas, USA, December 2002.

[15] M. Neubauer and P. Thiemann. From sequential programs to multi-tier applications by program transformation. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 221–232, New York, NY, USA, 2005. ACM Press.

[16] M. Pouzet. Lucid synchrone v3, release and reference manual. http://www.lri.fr/~pouzet/lucid-synchrone.

[17] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, 1987.

[18] É. Rutten. A framework for using discrete control synthesis in safe robotic programming and teleoperation. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'2001)*, pages 4104–4109, Seoul, Korea, May 2001.

[19] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3), 1992.

A Type system

$$(\textsc{Imm})\ H \vdash i : b\ \texttt{at}\ s/\{s\}/\emptyset$$

$$(\textsc{Inst})\ \frac{t \leq (H(x)) \qquad s \in \text{locations}(t)}{H \vdash x : t/\text{locations}(t)/\emptyset}$$

$$(\textsc{Comm})\ \frac{H \vdash e : t\ \texttt{at}\ s/\ell/T}{H \vdash e : t\ \texttt{at}\ s'/\ell \cup \{s'\}/T, [s \overset{c}{\mapsto} s']}$$

$$(\textsc{Pair})\ \frac{H \vdash e_1 : t_1/\ell_1/T_1 \qquad H \vdash e_2 : t_2/\ell_2/T_2}{H \vdash e_1, e_2 : t_1 \times t_2/\ell_1 \cup \ell_2/T_1, T_2}$$

$$(\textsc{Def})\ \frac{H \vdash e : t_1 \times \ldots \times t_n/\ell/T}{H \vdash (x_1, \ldots, x_n) = e : [t_1/x_1, \ldots, t_n/x_n]/\ell/T}$$

$$(\textsc{And})\ \frac{H \vdash D_1 : H_1/\ell_1/T_1 \qquad H \vdash D_2 : H_2/\ell_2/T_2}{H \vdash D_1\ \texttt{and}\ D_2 : H_1, H_2/\ell_1 \cup \ell_2/T_1, T_2}$$

$$(\textsc{Node})\ \frac{H, x_i : t_i, H_1 \vdash D : H_1/\ell_1/T_1 \qquad H, x_i : t_i, H_1 \vdash e : t/\ell_2/T_2}{\begin{array}{c} H \vdash \texttt{node}\ f(x_1, \ldots, x_n) = e\ \texttt{with}\ D : \\ [\text{gen}_H((t_1 \times \ldots \times t_n) \multimap\langle \ell_1/T_1, T_2\rangle\mapsto t)/f]/\ell_1 \cup \ell_2/\emptyset \end{array}}$$

$$(\textsc{App})\ \frac{\begin{array}{c} H \vdash f : t \multimap\langle \ell_1/T_1\rangle\mapsto (t_1 \times \ldots \times t_n)/\ell_2/T_2 s f'/D_1 \\ H \vdash e : t/\ell_3/T_3 s e'/D_2 \qquad T_1' \cong T_1 \end{array}}{H \vdash x = f(e) : [t_1/x_1, \ldots, t_n/x_n]/\ell_1 \cup \ell_2 \cup \ell_3/T_1', T_2, T_3}$$

$$(\textsc{Trans})\ \frac{H \vdash e : t\ \texttt{at}\ A_b/\ell_1/T_1 \qquad H \vdash u : \ell_2/T_2}{H \vdash\ \texttt{until}\ e\ \texttt{then}\ S\ u : \ell_1 \cup \ell_2/T_1, T_2, [A_b \overset{c}{\mapsto} A_d]}$$

$$(\textsc{Handler})\ \frac{H \vdash D : H'/\ell_1/T_1 \qquad H \vdash u : \ell_2/T_2}{H \vdash D\ u : H'/\ell_1 \cup \ell_2/T_1, T_2}$$

$$(\textsc{Automaton})\ \frac{\begin{array}{c} i \in \{1, \ldots, n\}, H \vdash h_i : H_i/\ell_i/T_i \\ \ell = \ell_1 \cup \ldots \cup \ell_n \qquad T = T_1, \ldots, T_n \qquad H' = \text{merge}(H_1, \ldots, H_n) \end{array}}{H \vdash \texttt{automaton}\ S_1 \to h_1 \ldots S_n \to h_n\ \texttt{end} : H'/\ell/T}$$