INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team Espresso

Environnement de spécification de programmes réactifs synchrones

Rennes

THEME COM

Activity Report

2004

Table of contents

1. Team **1**
2. Overall Objectives **1**
2.1. Introduction 1
2.2. Context and motivations 2
2.3. The polychronous approach 2
3. Scientific Foundations **3**
3.1.1. A synchronous model of computation 3
3.1.2. Declarative design languages 5
3.1.3. Compilation of Signal 7
3.1.3.1. Synchronization and scheduling analysis 7
3.1.3.2. Hierarchization 7
3.1.3.3. Example 8
3.1.3.4. Certification 10
4. Application Domains **10**
5. Software **10**
5.1. The Polychrony workbench 10
5.2. The APEX RTOS library 12
5.3. Real-time Java plug-in 13
5.4. A model of Signal in Coq 13
6. New Results **14**
6.1. A methodology for embedded software modeling and validation 14
6.1.1. A functional application domain. 14
6.1.2. Context. 15
6.1.3. Related work. 16
6.2. Embedded software modeling as behavioral type inference 16
6.2.1. Rationale. 16
6.2.2. Behavioral types. 17
6.2.3. Behavioral type inference. 18
6.3. Applications of behavioral modeling 19
6.3.1. Module checking. 19
6.3.2. Conformance checking. 20
6.3.3. Rethreading. 20
6.3.4. Modular design checking by components abstraction. 21
6.4. A UML profile for Real-Time and Embedded Systems 22
6.5. Behavioral inheritance in a synchronous model of computation 22
6.5.1. Example 22
6.6. Modeling and validation of asynchronous systems in synchronous frameworks 24
6.7. A compositional design methodology for refinement checking 25
6.8. Polychronous Modeling and Evaluation of Real-Time Systems 26
6.9. New features in Polychrony 27
7. Contracts and Grants with Industry **28**
7.1. Carroll project Protes (10/2003-10/2005) 28
7.2. Network of excellence Artist 28
7.3. Network of excellence Artist2 28
8. Other Grants and Activities **28**
8.1. INRIA associated projects program 28
9. Dissemination **30**

9.1. Advisory 30
9.2. Conferences 30
9.3. Events 30
9.4. Thesis 30
9.5. Teaching 31
9.6. Visits 31
10. Bibliography 31

1. Team

Team leader (INRIA)
Jean-Pierre Talpin [Research scientist (CR)]

Administrative assistant
Maryse Auffray [AA INRIA]

Staff members (INRIA)
Thierry Gautier [Research scientist (CR)]
Paul Le Guernic [Research director (DR)]

Staff member (CNRS)
Loïc Besnard [Research Engineer (IR)]

Faculty members
Bernard Houssais [Associate Professor (Université de Rennes 1), until Sept. 1st.]
Mickaël Kerbœuf [Lecturer (INSA), until Sept. 1st.]
Abdoulaye Gamatié [Lecturer (IFSIC)]

Ph. D. students
David Berner [INRIA]
Julien Ouy [INRIA, starting Oct. 1st.]

2. Overall Objectives

2.1. Introduction

The Espresso project-team builds upon the achievements of the former EPATR project-team to propose models, methods and tools for embedded system design.

- The model considered by the project-team is polychrony [8]. It is based on the paradigm of the synchronous hypothesis and allow for the specification of multi-clocked systems.
- The methods considered by the project-team put this model to work for the refinement-based (top-down) and component-based (bottom-up) design of embedded systems using correctness-preserving model transformations.
- The project-team makes a continuous effort to develop the Polychrony toolbox, freely available at http://www.irisa.fr/espresso/Polychrony.

Polychrony is an integrated development environment and technology demonstrator consisting of a compiler, a visual editor and a model checker. It provides a unified model-driven environment to perform embedded system design exploration by using top-down and bottom-up design methodologies formally supported by design model transformations from specification to implementation and from synchrony to asynchrony.

The company TNI-Valiosys supplies its commercial implementation, RT-Builder, used for large-scale industrial scale projects by Snecma/Hispano-Suiza and EADS – Airbus Industries (see http://www.tni-valiosys.com). Past and present collaborators of project-team Espresso through European, French and bilateral collaborations include CS-SI, CEA-List, MBDA, AONIX, SILICOMP, THALES, EDF, AIRBUS, VERIMAG, CEA.

2.2. Context and motivations

High-level embedded system design has gained prominence in the face of rising technological complexity, increasing performance requirements and shortening time to market demands for electronic equipments. Today, the installed base of intellectual property (IP) further stresses the requirements for adapting existing components with new services within complex integrated architectures, calling for appropriate mathematical models and methodological approaches to that purpose.

Over the past decade, numerous programming models, languages, tools and frameworks have been proposed to design, simulate and validate heterogeneous systems within abstract and rigorously defined mathematical models. Formal design frameworks provide well-defined mathematical models that yield a rigorous methodological support for the trusted design, automatic validation, and systematic test-case generation of systems.

However, they are usually not amenable to direct engineering use nor seem to satisfy the present industrial demand. As a matter of fact, the attention of the industry tends to shift to modeling frameworks based on general-purpose programming language variants, in response to a growing industry demand for higher abstraction-levels in the system design process and an attempt to fill the so-called *productivity gap*.

At present, a possibility of widening divergences between formal methods and industrial practices is perceivable. It seems that any useful methodology cannot avoid the industrial trend of using emerging programming languages. This contrasted picture calls for an effort toward the convergence between the theory of formal methods and the industrial practice and trends in system design.

Project-team Espresso aims at this convergence by considering the formal modeling framework of the Polychrony toolbox to serve as pivot formalism to import, transform, validate and export heterogeneous formalisms and languages.

2.3. The polychronous approach

Despite overwhelming advances in embedded systems design, existing techniques and tools merely provide *ad-hoc* solutions to the challenging issue of the productivity gap. The pressing demand for design tools has sometimes hidden the need to lay mathematical foundations below design languages. Many illustrating examples can be found, e.g. the variety of very different formal semantics found in state-diagram formalisms. Even though these design languages benefit from decades of programming practice, they still give rise to some diverging interpretations of their semantics.

The need for higher abstraction-levels and the rise of stronger market constraints now make the need for unambiguous design models more obvious. This challenge requires models and methods to translate a high-level system specification into a distribution of purely sequential programs and to implement semantics-preserving transformations and high-level optimizations such as hierarchization (sequentialization) or desynchronization (protocol synthesis).

In this aim, system design based on the so-called “synchronous hypothesis” has focused the attention of many academic and industrial actors. The synchronous paradigm consists of abstracting the non-functional implementation details of a system and lets one benefit from a focused reasoning on the logics behind the instants at which the system functionalities should be secured.

With this point of view, synchronous design models and languages provide intuitive models for embedded systems [3]. This affinity explains the ease of generating systems and architectures and verify their functionalities using compilers and related tools that implement this approach.

In the relational mathematical model behind the design language Signal, the supportive data-flow notation of Polychrony, this affinity goes beyond the domain of purely sequential systems and synchronous circuits and embraces the context of complex architectures consisting of synchronous circuits and desynchronization protocols: globally asynchronous and locally synchronous architectures (GALS).

This unique feature is obtained thanks to the fundamental notion of *polychrony*: the capability to describe systems in which components obey to multiple clock rates. It provides a mathematical foundation to a notion of *refinement*: the ability to model a system from the early stages of its requirement specifications (relations, properties) to the late stages of its synthesis and deployment (functions, automata).

The notion of polychrony goes beyond the usual scope of a programming language, allowing for specifications and properties to be described. As a result, the Signal design methodology draws a continuum from synchrony to asynchrony, from specification to implementation, from abstraction to refinement, from interface to implementation. SIGNAL gives the opportunity to seamlessly model embedded systems at multiple levels of abstraction while reasoning within a simple and formally defined mathematical model.

The inherent flexibility of the abstract notion of signal handled in Signal invites and favors the design of correct-by-construction systems by means of well-defined model transformations that preserve the intended semantics and stated properties of the architecture under design.

3. Scientific Foundations

Embedded systems are not new, but their pervasive introduction in ordinary-life objects (cars, telephone, home appliances) brought a new focus onto design methods for such systems. New development techniques are needed to meet the challenges of productivity in a competitive environment. Synchronous languages rely on the *synchronous hypothesis*, which lets computations and behaviors be divided into a discrete sequence of *computation steps* which are equivalently called *reactions* or *execution instants*. In itself this assumption is rather common in practical embedded system design. But the synchronous hypothesis adds to this the fact that, *inside each instant*, the behavioral propagation is well-behaved (causal), so that the status of every signal or variable is established and defined prior to being tested or used. This criterion, which may be seen at first as an isolated technical requirement, is in fact the key point of the approach. It ensures strong semantic soundness by allowing universally recognized mathematical models to be used as supporting foundations. In turn, these models give access to a large corpus of efficient optimization, compilation, and formal verification techniques. The synchronous hypothesis also guarantees full equivalence between various levels of representation, thereby avoiding altogether the pitfalls of non-synthesizability of other similar formalisms. In that sense the synchronous hypothesis is, in our view, a major contribution to the goal of *model-based design* of embedded systems.

We shall describe the synchronous hypothesis and its mathematical background, together with a range of design techniques enpowered by the approach. Declarative formalisms implementing the synchronous hypothesis can be cast into a model of computation (proposed in [8]) consisting of a *domain* of traces or behaviors and of semi-lattice structure that renders the synchronous hypothesis using a timing equivalence relation: clock equivalence. Asynchrony can be superimposed on this model by considering a flow equivalence relation as well as heterogeneous systems [31] by parameterizing composition with arbitrary timing relations.

3.1.1. A synchronous model of computation

We consider a partially-ordered set of tags t to denote instants seen as symbolic periods in time during which a reaction takes place. The relation $t_1 \leq t_2$ says that t_1 occurs before t_2. Its minimum is noted 0. A totally ordered set of tags C is called a *chain* and denotes the sampling of a possibly continuous or dense signal over a countable series of causally related tags. Events, signals, behaviors and processes are defined as follows:

- an *event* e is a pair consisting of a value v and a tag t,
- a *signal* s is a function from a *chain* of tags to a set of values.
- a *behavior* b is a function from a set of names x to signals.
- a *process* p is a set of behaviors that have the same domain.

In the remainder, we write $\mathrm{tags}(s)$ for the tags of a signal s, $\mathrm{vars}(b)$ for the domains of b, $b|_X$ for the projection of a behavior b on a set of names X and b/X for its complementary. Figure 1 depicts a behavior b over three signals named x, y and z. Two frames depict timing domains formalized by chains of tags. Signals x and y belong to the same timing domain: x is a down-sampling of y. Its events are synchronous to odd occurrences of events along y and share the same tags, e.g. t_1. Even tags of y, e.g. t_2, are ordered along its chain, e.g. $t_1 < t_2$, but absent from x. Signal z belongs to a different timing domain. Its tags, e.g. t_3 are not ordered with respect to the chain of y, e.g. $t_1 \neg \leq t_3$ and $t_3 \neg \leq t_1$.

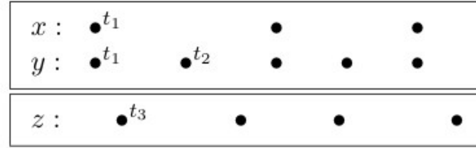

Figure 1. Behavior b over three signals x, y and z in two clock domains

Synchronous composition is noted $p \| q$ and defined by the union $b \cup c$ of all behaviors b (from p) and c (from q) which hold the same values at the same tags $b|_I = c|_I$ for all signal $x \in I = \text{vars}(b) \cap \text{vars}(c)$ they share. Figure 2 depicts the synchronous composition (Figure 2, right) of the behaviors b (Figure 2, left) and the behavior c (Figure 2, middle). The signal y, shared by b and c, carries the same tags and the same values in both b and c. Hence, $b \cup c$ defines the synchronous composition of b and c.

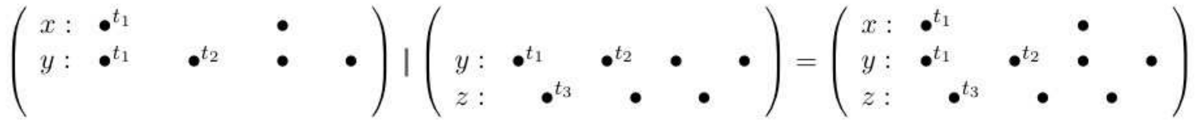

Figure 2. Synchronous composition of $b \in p$ and $c \in q$

A scheduling structure is defined to schedule the occurrence of events along signals during an instant t. A scheduling $\rightarrow$ is a pre-order relation between dates x_t where t represents the time and x the location of the event. Figure 3 depicts such a relation superimposed to the signals x and y of Figure 1. The relation $y_{t_1} \rightarrow x_{t_1}$, for instance, requires y to be calculated before x at the instant t_1. Naturally, scheduling is contained in time: if $t < t'$ then $x_t \rightarrow^b x_{t'}$ for any x and b and if $x_t \rightarrow^b x_{t'}$ then $t' \neg < t$.

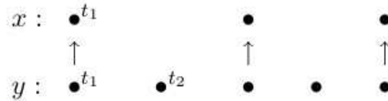

Figure 3. Scheduling relations between simultaneous events

A synchronous structure is defined by a semi-lattice structure to denote behaviors that have the same timing structure. The intuition behind this relation is depicted in Figure 4. It is to consider a signal as an elastic with ordered marks on it (tags). If the elastic is stretched, marks remain in the same relative (partial) order but have more space (time) between each other. The same holds for a set of elastics: a behavior. If elastics are equally stretched, the order between marks is unchanged. In the figure 4, the time scale of x and y changes but the partial timing and scheduling relations are preserved. Stretching is a partial-order relation which defines

clock equivalence. Formally, a behavior c is a *stretching* of b of same domain, written $b \leq c$, iff there exists an increasing bijection on tags f that preserves the timing and scheduling relations. If so, c is the image of b by f. Last, the behaviors b and c are said *clock-equivalent*, written $b \sim c$, iff there exists a behavior d s.t. $d \leq b$ and $d \leq c$.

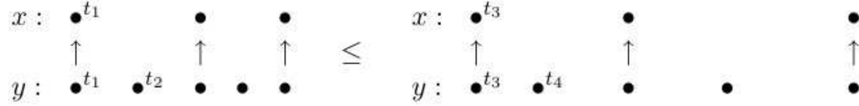

Figure 4. Relating synchronous behaviors by stretching.

3.1.2. Declarative design languages

Signal [4] is a declarative design language expressed within the polychronous model of computation. In Signal, a process P is an infinite loop that consists of the synchronous composition $P|Q$ of simultaneous equations $x = y\ f\ z$ over signals named x, y, z. The restriction of a signal name x to a process P is noted P/x.

$$P, Q ::= x = y\ f\ z \ \mid\ P/x \ \mid\ P|Q$$

Equations $x = y\ f\ z$ in Signal more generally denote processes that define timing relations between input and output signals. There are four primitive combinators in Signal:

- delay $x = y\$1\ \mathtt{init}\ v$, initially defines the signal x by the value v and then by the previous value of the signal y. The signal y and its delayed copy $x = y\$1\ \mathtt{init}\ v$ are synchronous: they share the same set of tags $t_1, t_2, \cdots$. Initially, at t_1, the signal x takes the declared value v and then, at tag t_n, the value of y at tag t_{n-1}.

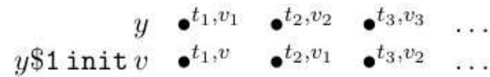

Figure 5.

- sampling $x = y\ \mathtt{when}\ z$, defines x by y when z is true (and both y and z are present); x is present with the value v_2 at t_2 only if y is present with v_2 at t_2 and if z is present at t_2 with the value true. When this is the case, one needs to schedule the calculation of y and z before x, as depicted by $y_{t_2} \rightarrow x_{t_2} \longleftarrow z_{t_2}$.
- merge $x = y\ \mathtt{default}\ z$, defines x by y when y is present and by z otherwise. If y is absent and z present with v_1 at t_1 then x holds (t_1, v_1). If y is present (at t_2 or t_3) then x holds its value whether z is present (at t_2) or not (at t_3).

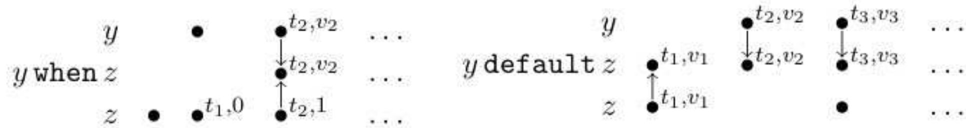

Figure 6.

The structuring element of a Signal specification is a process. A process accepts input signals originating from possibly different clock domains to produce output signals when needed. This allows, for instance, to specify a counter where the inputs `tick` and `reset` and the output `value` have independent clocks. The body of `counter` consists of one equation that defines the output signal `value`. Upon the event `reset`, it sets the count to 0. Otherwise, upon a `tick` event, it increments the count by referring to the previous value of `value` and adding 1 to it. Otherwise, if the count is solicited in the context of the counter process (meaning that its clock is active), the counter just returns the previous count without having to obtain a value from the `tick` and `reset` signals.

```
process counter = (? event tick, reset ! integer value)
     (| value := (0 when reset)
         default ((value$ init 0 + 1) when tick)
         default (value$ init 0)
      |);
```

Figure 7.

A Signal process is a structuring element akin to a hierarchical block diagram. A process may structurally contain sub-processes. A process is a generic structuring element that can be specialized to the timing context of its call. For instance, the definition of a synchronized counter starting from the previous specification consists of its refinement with synchronization. The input tick and reset clocks expected by the process `counter` are sampled from the boolean input signals `tick` and `reset` by using the `when tick` and `when reset` expressions. The count is then synchronized to the inputs by the equation `reset ^= tick ^= count`.

```
process synccounter = (? boolean tick, reset ! integer value)
     (| value := counter (when tick, when reset)
      | reset ^= tick ^= value
      |);
```

Figure 8.

3.1.3. Compilation of Signal

Sequential code generation starting from a Signal specification starts with an analysis of its implicit synchronization and scheduling relations. This analysis yields the control and data flow graphs that define the class of sequentially executable specifications and allow to generate code.

3.1.3.1. Synchronization and scheduling analysis

In SIGNAL, the clock $\hat{}\,x$ of a signal x denotes the set of instants at which the signal x is present. It is represented by a signal that is true when x is present and that is absent otherwise. Clock expressions represent control. The clock $\texttt{when}\,x$ (resp. $\texttt{when not}\,x$) represents the time tags at which a boolean signal x is present and true (resp. false). The empty clock is written 0 and clocks expressions e combined using conjunction, disjunction and symmetric difference. Clocks equations E are Signal processes: the equation $e\ \hat{}=\ e'$ synchronizes the clocks e and e' while $e\ \hat{}<\ e'$ specifies the containment of e in e'. Additionally, explicit scheduling relations $x \to y\ \texttt{when}\ e$ allow to schedule the calculation of signals (e.g. x after y at the clock e).

$$
\begin{array}{lcll}
e & ::= & \hat{}x \mid \texttt{when}\,x \mid \texttt{when not}\,x \mid e\,\hat{}{+}\,e' \mid e\,\hat{}{-}\,e' \mid e\,\hat{}{*}\,e' \mid 0 & \text{(clock expression)} \\
E & ::= & () \mid e\hat{}{=}e' \mid e\hat{}{<}e' \mid x \to y\,\texttt{when}\,e \mid E|E' \mid E/x & \text{(clock relations)}
\end{array}
$$

Figure 9.

Any Signal process P nicely corresponds to a system of clock relations E that denotes its timing and scheduling structure. It can be defined by the inference system $P : E$ of Figure 10.

$$
\begin{array}{c}
x := y\$1\ \texttt{init}\ v : \hat{}x\hat{}{=}\hat{}y \\
x := y\ \texttt{when}\ z : \hat{}x\hat{}{=}\hat{}y\ \texttt{when}\ z \mid y \to x\ \texttt{when}\ z \\
x := y\ \texttt{default}\ z : \hat{}x\hat{}{=}\hat{}y\,\hat{}{+}\,\hat{}z \mid y \to x \mid z \to x\ \texttt{when}\ (\hat{}z\,\hat{}{-}\,\hat{}y)
\end{array}
\qquad
\frac{P : E \quad Q : E'}{P|Q : E|E'}
\qquad
\frac{P : E}{P/x : E/x}
$$

Figure 10. Clock inference system

3.1.3.2. Hierarchization

The clock and scheduling relations E of a process P define the control-flow and data-flow graphs that hold all necessary information to compile a Signal specification upon satisfaction of the property of *endochrony*, illustrated in Figure 11. A process is said endochronous iff, given a set of input signals (x and y in Figure 11) and flow-equivalent input behaviors (datagrams on the left in Figure 11), it has the capability to reconstruct a unique synchronous behavior up to clock-equivalence: the datagrams of the input signals in the middle of Figure 11 and of the output signal on the right of Figure 11 are ordered in clock-equivalent ways.

To determine the order $x \preceq y$ in which signals are processed during the period of a reaction, clock relations E play an essential role. The process of determining this order is called hierarchization and consists of an insertion algorithm which proceeds in three easy steps:

1. First, equivalence classes are defined between signals of same clock: if $E \Rightarrow\ \hat{}\,x\ \hat{}=\ \hat{}\,y$ then $x \preceq y$ (we write $E \Rightarrow E'$ iff E implies E').

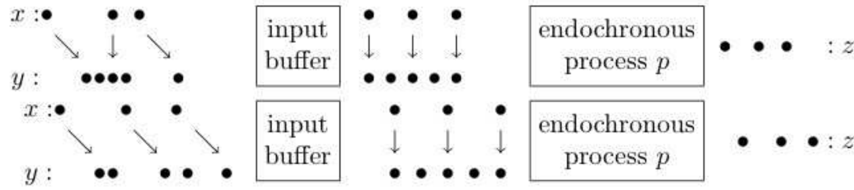

Figure 11. Endochrony: from flow-equivalent inputs to clock-equivalent outputs

2. Second, elementary partial order relations are constructed between sampled signals: if $E \Rightarrow$ `^x ^= when y` or $E \Rightarrow$ `^x ^= when not y` then $y \preceq x$.
3. Last, assume a partial order of maximum z such that $E \Rightarrow$ `^z = ^y f ^w` (for some $f \in$ {`^+`, `^*`, `^-`}) and a signal x such that $y \preceq x \succeq w$, then insertion consists of attaching z to x by $x \preceq z$.

The insertion algorithm proposed in [1] yields a canonical representation of the partial order $\preceq$ by observing that there exists a unique minimum clock x below z such that rule 3 holds. Based on the order $\preceq$, one can decide whether E is *hierarchical* by checking that its clock relation $\preceq$ has a minimum, written $\min_{\preceq} E \in \mathrm{vars}(E)$, so that $\forall x \in \mathrm{vars}(E), \exists y \in \mathrm{vars}(E), y \preceq x$. If E is furthermore *acyclic* (i.e. $E \Rightarrow x \rightarrow x$ `when` e implies $E \Rightarrow e$ `^=` 0, for all $x \in \mathrm{vars}(E)$) then the analyzed process is endochronous, as shown in [8].

3.1.3.3. Example

The implications of hierarchization for code generation can be outlined by considering the specification of a one-place buffer in Signal (Figure 14, left). Process `buffer` implements two functionalities. One is the process `alternate` which desynchronizes the signals `i` and `o` by synchronizing them to the true and false values of an alternating boolean signal `b`. The other functionality is the process `current`. It defines a `cell` in which values are stored at the input clock `^i` and loaded at the output clock `^o`. `cell` is a predefined Signal operation defined by: Clock inference (Figure 14, middle) applies the clock inference system of Figure 10 to the process

$$x := y \ \texttt{cell} \ z \ \texttt{init} \ v =^{def} (m := x\$1 \ \texttt{init} \ v \,|\, x := y \ \texttt{default} \ m \,|\, \hat{}x \hat{}= \hat{}y \hat{}+ \hat{}z) \,/\, m$$

Figure 12.

`buffer` to determine three synchronization classes. We observe that `b, c_b, zb, zo` are synchronous and define the master clock synchronization class of `buffer`. There are two other synchronization classes, `c_i` and `c_o`, that corresponds to the true and false values of the boolean flip-flop variable `b`, respectively :

This defines three nodes in the control-flow graph of the generated code, Figure 14, right. At the main clock `c_b`, `b` and `c_o` are calculated from `zb`. At the sub-clock `b`, the input signal `i` is read. At the sub-clock `c_o` the output signal `o` is written. Finally, `zb` is determined. Notice that the sequence of instructions follows the scheduling relations determined during clock inference.

Whereas Signal uses a hierarchization algorithm to find a sequential execution path starting from a system of clock relations, Lustre leaves this task to engineers, which must provide a sound, fully synchronized program in the first place: well-synchronized Lustre programs correspond to hierarchized Signal specifications.

$$\mathtt{b} \multimap\!\!\!\diamond\!\!\!\multimap \mathtt{c_b} \multimap\!\!\!\diamond\!\!\!\multimap \mathtt{zb} \multimap\!\!\!\diamond\!\!\!\multimap \mathtt{zo} \text{ and } \mathtt{b} \preceq \mathtt{c_i} \multimap\!\!\!\diamond\!\!\!\multimap \mathtt{i} \text{ and } \mathtt{b} \preceq \mathtt{c_o} \multimap\!\!\!\diamond\!\!\!\multimap \mathtt{o}$$

Figure 13.

```
process buffer = (? i ! o)
  (| alternate (i, o)
   | o := current (i)
   |) where
process alternate = (? i, o ! )
  (| zb := b$1 init true
   | b := not zb
   | o ^= when not b
   | i ^= when b
   |) / b, zb;
process current = (? i ! o)
  (| zo := i cell ^o init false
   | o  := zo when ^o
   |) / zo;
```

```
(| c_b ^= b
 | b   ^= zb
 | zb  ^= zo
 | c_i := when b
 | c_i ^= i
 | c_o := when not b
 | c_o ^= o
 | i -> zo when ^i
 | zb -> b
 | zo -> o when ^o
 |) / zb, zo, c_b,
       c_o, c_i, b;
```

```
buffer_iterate () {
  b = !zb;
  c_o = !b;
  if (b) {
    if (!r_buffer_i(&i))
      return FALSE;
  }
  if (c_o) {
    o = i;
    w_buffer_o(o);
  }
  zb = b;
  return TRUE;
}
```

Figure 14. Specification, clock analysis and code generation in Signal

3.1.3.4. Certification

The simplicity of the single-clocked model of Lustre eases program analysis and code generation and its commercial implementation, Scade by Esterel Technologies, provides a certified C code generator. Its combination to Sildex, the commercial implementation of Signal by TNI-Valiosys, as a front-end for architecture mapping and early requirement specification is the methodology advocated in the IST project Safeair (URL: http://www.safeair.org). The formal validation and certification of synchronous program properties has been the subject of numerous studies. In [46], a co-inductive axiomatization of Signal in the proof assistant Coq [36], based on the calculus of constructions [53], is proposed.

The application of this model is two-folds. It allows, first of all, for the exhaustive verification of formal properties of infinite-state systems. Two case studies have been developed. In [42], a faithful model of the steam-boiler problem was given in Signal and its properties proved with Signal's Coq model. It is applied to proving the correctness of real-time properties of a protocol for loosely time-triggered architectures, extending previous work proving the correctness of its finite-state approximation [41].

Another and important application of modeling Signal in the proof assistant Coq is being explored and consists of developing a reference compiler translating Signal programs into Coq assertion. This translation allows to represent model transformations performed by the Signal compiler as correctness preserving transformations of Coq assertions, yielding a costly yet correct-by-construction synthesis of target code.

Other approaches to the certification of generated code have been investigated. In [47], validation is achieved by checking a model of the C code generated by the Signal compiler in the theorem prover PVS with respect to a model of its source specification: translation validation.

4. Application Domains

The application domains covered by the Polychrony toolbox are engineering areas where a system design-flow requires high-level model transformations and verifications to be applied during the development-cycle.

The project-team has focused on developing such integrated design methods in the context of avionics applications, through the European IST projects Sacres, Syrf, Safeair. This research track is being continued in the submitted Espace (*avionics*) and Sea (*automotive*) projects.

In this context, Polychrony is seen as a platform on which the architecture of an embedded system can be specified from the earliest design stages until the late deployment stages through a number of formally verifiable design refinements.

Recent trends in *system-level design* show, in a far from unrelated way, the need for modeling systems on chips as globally asynchronous and locally synchronous systems. It is indeed manifest in the charter of the ACM-IEEE MEMOCODE conference [13]. It is the subject of an ongoing collaboration of project-team Espresso with UC San Diego and Virginia Tech through INRIA associate-projects program.

5. Software

5.1. The Polychrony workbench

Participants: Loïc Besnard, Thierry Gautier, Paul Le Guernic.

Polychrony is an integrated development environment and technology demonstrator consisting of a compiler, of a visual editor and of a model checker. It provides a unified model-driven environment to perform embedded system design exploration by using top-down and bottom-up design methodologies formally supported by design model transformations from specification to implementation and from synchrony to asynchrony.

Polychrony supports the synchronous, multi-clocked, dataflow specification language Signal. It is being extended by plugins to capture SystemC modules or real-time Java classes within the workbench. It allows to perform validation and verification tasks, e.g., with the integrated SIGALI model checker, the Coq theorem prover, or with the Spin model checker. Polychrony is registered at the APP and is freely distributed from

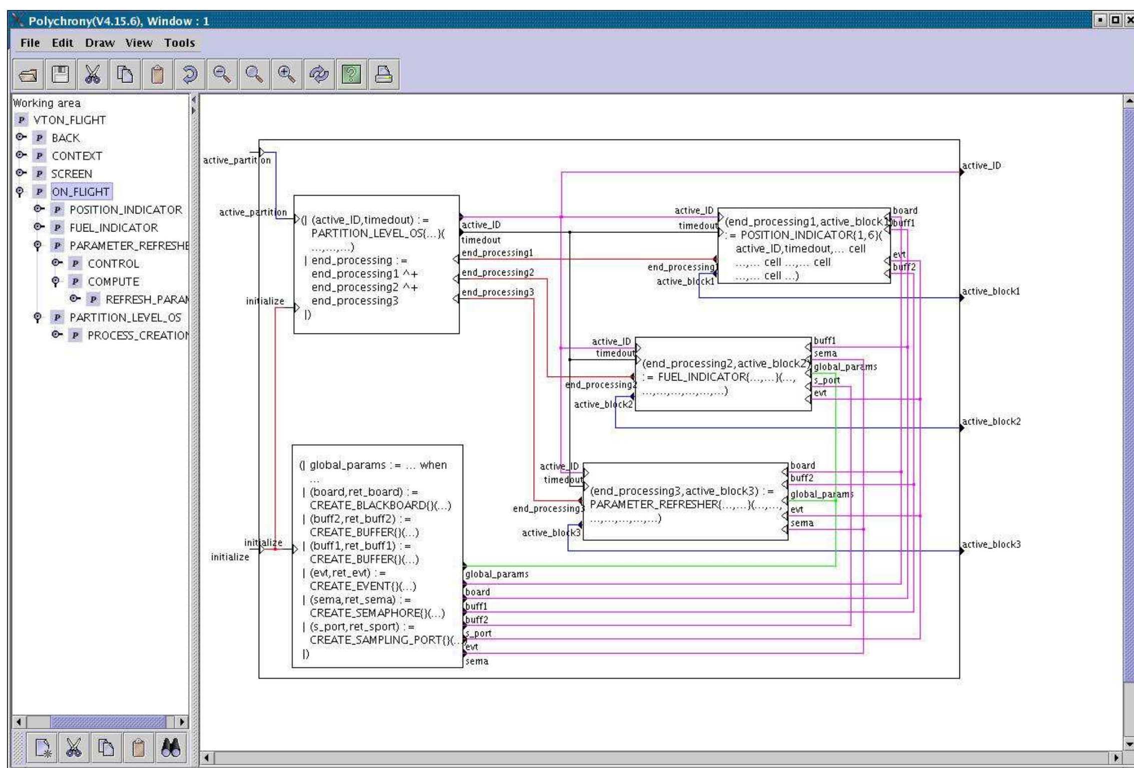

Figure 15. Avionics application modeling using the visual editor of the Polychrony workbench

http://www.irisa.fr/espresso/Polychrony for non-commercial use. Based on the Signal language, it provides a formal framework:

1. to validate a design at different levels,
2. to refine descriptions in a top-down approach,
3. to abstract properties needed for black-box composition,
4. to assemble predefined components (bottom-up with COTS).

The company TNI-Valiosys supplies a commercial implementation of Polychrony, called RT-Builder, used for industrial scale projects by Snecma/Hispano-Suiza and Airbus Industries (see ://www.tni-valiosys.com).

Polychrony is a set of tools composed of:

1. A Signal batch compiler providing a set of functionalities viewed as a set of services for, e.g., program transformations, optimizations, formal verification, abstraction, separate compilation, mapping, code generation, simulation, temporal profiling, etc.
2. A GUI with interactive access to compiling functionalities.
3. The SIGALI tool, an associated formal system for formal verification and controller synthesis, jointly developed with the Vertecs project-team (http://www.irisa.fr/vertecs).

Polychrony offers services for modeling application programs and architectures starting from high-level and heterogeneous input notations and formalisms. These models are imported in Polychrony using the data-flow notation Signal. Polychrony operates these models by performing global transformations and optimizations on them (hierarchization of control, desynchronization protocol synthesis, separate compilation, clustering, abstraction) in order to deploy them on mission specific target architectures. C, C++, multi-threaded and real-time Java and SynDex code generators are provided. The connection to the SynDEx distribution tool (http://www-rocq.inria.fr/syndex) has been developed in the context of the RNTL project Acotris.

5.2. The APEX RTOS library

Participants: Abdoulaye Gamatié, Thierry Gautier.

The Apex interface, defined in the ARINC standard [26], provides an avionics application software with the set of basic services to access the operating-system and other system-specific resources. Its definition relies on the Integrated Modular Avionics approach (IMA, [27]). A main feature in an IMA architecture is that several avionics applications (possibly with different critical levels) can be hosted on a single, shared computer system. Of course, a critical issue is to ensure safe allocation of shared computer resources in order to prevent fault propagations from one hosted application to another. This is addressed through a functional partitioning of the applications with respect to available time and memory resources. The allocation unit that results from this decomposition is the *partition*.

A partition is composed of *processes* which represent the executive units (an ARINC partition/process is akin to a Unix process/task). When a partition is activated, its owned processes run concurrently to perform the functions associated with the partition. The process scheduling policy is priority preemptive.

Each partition is allocated to a processor for a fixed time window within a major time frame maintained by the operating system. Suitable mechanisms and devices are provided for communication and synchronization between processes (e.g. *buffer*, *event*, *semaphore*) and partitions (e.g. *ports* and *channels*).

The specification of the ARINC 651-653 services in Signal is now part of the distribution Polychrony and offers a complete implementation of the Apex communication, synchronization, process management and partitioning services. Its Signal implementation consists of a library of generic, parameterizable Signal modules.

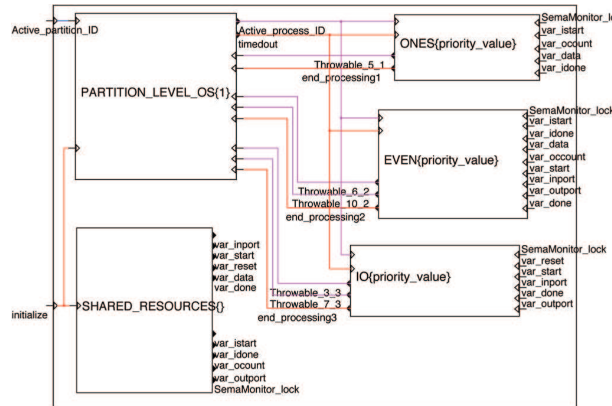

Figure 16.

5.3. Real-time Java plug-in

Participants: Loïc Besnard, Jean-Pierre Talpin.

The real-time Java plug-in of Polychrony is a prototype tool which was developed during the frame of the RNTL project Expresso. It consists of modeling a domain-specific subset of the real-time Java specification (for modeling avionics applications or control dominated embedded systems) in Signal: thus it provides a compiler from this subset of RT-Java into Signal. This model is obtained from a given Java class-file hierarchy (at either byte-code or source level) by, first, automatic modeling of the application architecture using instances of the Polychrony-Apex services and, second, by translating periodic/sporadic Java threads and event handlers in the data-flow polychronous design language Signal [49].

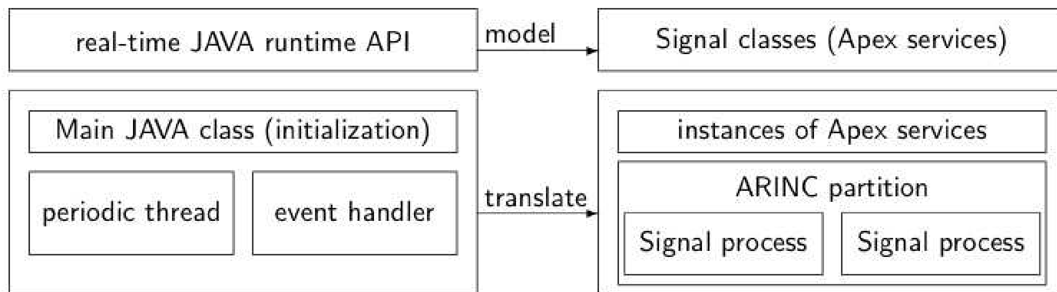

Figure 17. Architecture of the real-time Java plug-in for Polychrony

5.4. A model of Signal in Coq

Participants: Mickaël Kerbœuf, Jean-Pierre Talpin.

The verification of a reactive system is usually done by elaborating a *discrete* model of the system specified in a dedicated formalism and then by checking a property against the model. The use of formal proof systems

enables to prove *hybrid properties* about *infinite state systems*: the *correctness* and the *completeness* of a reactive system.

To this aim, the Espresso project-team has developed a complete model of the Signal design language in Coq [46]. More precisely, we have defined a translation scheme of the trace semantics of Signal to the logical framework of Coq. We have conducted several case studies to demonstrate the applicability of the approach to resolve sophisticated verification problems: a complete model and proof of the well-known steam-boiler problem [42], the correctness of an implementation of a Signal protocol for loosely timed-triggered architectures [41].

Such a proof, of course, cannot always be done automatically: it requires human-interaction to direct the proof strategy. The prover can nonetheless automate its most tedious and mechanical parts. In general, formal proofs of programs are difficult and time-consuming. In the particular case of modeling a reactive system using Signal, experience however shows that this difficulty is significantly reduced thanks to the combined declarative style of programming and a relational style of modeling.

6. New Results

6.1. A methodology for embedded software modeling and validation

Participants: David Berner, Abdoulaye Gamatié, Paul Le Guernic, Jean-Pierre Talpin.

The popular slogan "*write once, run anywhere*" effectively renders the expressive capabilities of general purpose programming languages for developing, deploying, and reusing target-independent applications. Generality and simplicity has driven most attention of the compiler technology community to developing local and compositional compiler optimization techniques. When it comes to the implementation of embedded software, this approach is however far from satisfactory, especially in hard real-time system design (e.g. airborne systems, digital circuits) where conformance to real-time specifications is critical. Domain-specific models and languages, such as those proposed under the synchronous programming paradigm, provide the necessary formal engineering models and design methodologies to allow for a program written once to be mapped on any distributed execution architecture by using global transformation and optimization techniques. Our aim is to relate this domain-specific model to embedded software development using general-purpose environments. To this end, we set the methodological framework of our synchronous model of computation within the general and reusable concept of a type system targeting the generic programming language setting of GCC's intermediate representations (three-address code and static single assignment). We give formal semantics to both our type system and the functional subset of SSA under consideration, define a type inference system and prove its correctness, before to depict the applications of our technique as developed in our project and presented in previous works.

6.1.1. A functional application domain.

We consider embedded software implemented by resource-constrained multi-threaded programs on a specific runtime sub-system (e.g., the real-time JVM, an RTOS, or simply hardware) which we call its execution architecture. Our technique consists of a type inference system that relates threads (imperative programs in intermediate form) to propositions expressed by synchronous transition systems that describe their behavior. Let us outline the extent of our technique by depicting a test-case studied in [49] (Figure, right). We consider modeling a real-time Java program consisting of three threads (Figure, right), a scheduler (Figure, top-left) and shared resources control (Figure, bottom-left). This decomposition is obtained by partitioning the executable program and its environment into:

- *the execution architecture*: a hardware platform, a middle-ware library, a real-time operating system, a virtual machine (e.g. in Java), a simulation kernel (e.g. in SystemC). The execution architecture describes an API of generic process and communication management services.

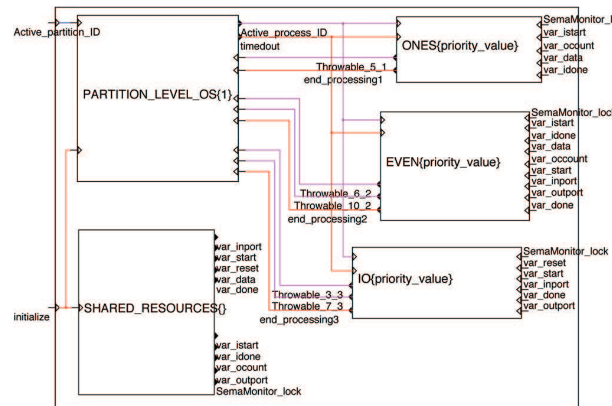

Figure 18.

- *the application architecture*: a program, starting from the `main()` procedure, which initializes and links objects to form a hierarchical structure of shared data and communicating threads. The application mapping constructively describes the architecture of the system.
- *the application functionalities*: a set of program threads which periodically or sporadically react to inputs from the environment by interacting with each other for the access to shared data.

Our methodology consists of considering the three elements of an embedded system (its execution and application architectures, its application functionalities) in specific ways.

- *modeling*: the execution architecture, viewed through an application programming interface (API) of generic services, is modeled by template propositions. For instance, the procedure for thread creation in an RTOS API corresponds to a template proposition in the RTOS model whose parameters are the number of threads supported by the application scheduler, the period and deadline of the thread (for a real-time thread), etc.
- *analysis*: the application architecture, viewed as a hierarchical structure, is interpreted to elaborate a model by the instantiation of generic API services to the parameters and initial values provided in the program (e.g. thread parameters).
- *translation*: each thread consists of a sequential program that describes a functionality to be periodically or sporadically executed by the scheduler and corresponds to a particular model.

This allows for a complete separation of the virtual (threading or functional) architecture of an application from its actual, real-time and resource-constrained implementation: it provides an implementation of the "*write once run anywhere*" slogan in embedded system design.

6.1.2. Context.

Our methodology arises from previous work on real-time operating systems modeling, embedded systems modeling and verification in the Polychrony workbench (URL: http://www.irisa.fr/espresso/Polychrony) a tool-set for embedded system design based on a multi-clocked synchronous model of computation and implemented by the data-flow notation Signal. In [35], the authors describe the implementation of a real-time operating system standard for avionics application: ARINC [35]. The commercial implementation of this library, RT-Builder from TNI-Valiosys, is used for industrial-scale embedded software engineering project in avionics. In [49], this model is used to describe key services of the real-time Java virutal machine. It is applied

to rethreading multi-threaded real-time Java programs by global optimization. In [18], the application of our methodology to system-level design is further developed by studying its application to checking behavioral conformance between embedded systems described in SpecC and at heterogeneous levels of abstraction. In [23], a generic translation scheme of SystemC programs to the Polychrony workbench is described by considering a static single assignment intermediate representation due to the GCC project [45]. It is applied to design checking (e.g. race and lock detection). In [19], it is applied to modular verification by model checking and component-wise model abstraction.

6.1.3. Related work.

A related direction of research is software model checking using the popular tools Bandera [38], Mops [34], Verisoft [37], Modex [40], Slam [30], CBMC [44], Magic [33], Blast [32], Java PathFinder [51]. Most software model checking (SMC) tools proceed by extracting temporal logic models of source programs (either Java or C but rarely both) and perform sophisticated and efficient abstractions to drastically accelerate property verification.

Our approach contrasts with the software model checking trend in that it is primarily aimed at *modeling* software and then perform either of global model transformations (desynchronization, rethreading, etc) and code generation [49], conformance checking by finite-flow equivalence using model checking techniques [18] or modular state-less abstraction for efficient property verification [19]. As such, our approach most closely relates to that of Modex [40] in which temporal property models are extracted for later verification with Spin [39]. We experienced that representing such models using executable specifications expressed in a multi-clocked synchronous model offers the additional benefit of operating correctness-preserving model transformations such as protocol synthesis (desynchronization [18]) or static scheduling (rethreading [49]). Finally, and unlike most related approaches in SMC, which are geared towards a particular programming language, we focus on a language-independent intermediate representation of Gnu's GCC.

We share the aim of a scalable and correct-by-construction exploration of abstraction-refinement of system behaviors with the work of Henzinger et al. on interface automata . Our approach primarily differs from interface automata in the data-structure used in the Polychrony workbench: clock equations, boolean propositions and state variable transitions express the multi-clocked synchronous behavior of a system.

Compared to an automata-based approach, our declarative approach allows to hierarchically explore abstraction capabilities and to cover design exploration with the methodological notion of refinement along the whole design cycle of the system, ranging from the early requirements specification to the latest sequential and distributed code-generation [8].

6.2. Embedded software modeling as behavioral type inference

Participants: David Berner, Paul Le Guernic, Jean-Pierre Talpin.

We set our methodological framework within the general paradigm of a behavioral type system that associates meaning to software functionalities. The type system is cast in the generic programming language-oriented context of the three-address code (TAC) and static single assignment (SSA) intermediate representations (IR) of GCC.

6.2.1. Rationale.

To allow for an easy grasp on the type system proposed for modeling behaviors, we outline the analysis of an imperative program, Figure 19, and depict the construction of its type, Figure 20. Figure 19 depicts a simple C code fragment consisting of an iterative program that counts the number of bits set to one in the variable `idata`. While `idata` is not equal to zero, it adds its right-most bit to an output count variable *ocount* and shifts it right in order to process the next bit. In the intermediate representation (IR) of the program (Figure 19, second column), all variables (`idata` and `ocount`) are read and written once per cycle.

This IR can equally be one of the TAC and SSA formats of GCC. Label `L2` is the entry point of the block associated with the while loop. The first instruction loads the input variable `idata` into the register `T1`. The second instruction stores the result of its comparison with 0 in the register `T0`. If `T0` is false, control is passed

```
while (idata != 0) {                      L2:T1 = idata;              | T3 = T1 & 1;
   ocount = ocount + (idata & 1);            T0 = T1 == 0;            | ocount = T2 + T3;
   idata = idata >> 1;                       if T0 then goto L3;      | idata = T1 >> 1;
}                                            T2 = ocount;             | goto L2;
```

Figure 19. From a C-like program to its intermediate representation

to block L3. Otherwise, the next instruction is executed: the variable ocount is loaded into T2, the last bit of T1 is loaded into T3, the sum of T2 and T3 assigned to ocount and the right-shift of T1 assigned to idata. The block terminates with an unconditional branch back to label L2.

6.2.2. Behavioral types.

The meaning of this C program fragment is given in a minimalist formalism akin to Pnueli's synchronous transition systems (STS, [47]). It not only describes a behavior of the program suitable for its formal verification but also allows for global model transformations to be performed on it. Let us zoom on the block L2 in the example of Figure 20. The behavioral type of the block L2 consists of the simultaneous composition of logical propositions that form a synchronous transition system. Each proposition is associated with one instruction: it specifies its *invariants*: it tells when the instruction is executed, what it computes, when it passes control to the next statement, when it branches to another block.

```
L2:T1 = idata;           | L2⇒T1  :=idata
   T0 = T1 == 0;         |      T0  :=(T1 = 0)
   if T0 then goto L3;   |      T0  ⇒L3′
   T2 = ocount;          |      ¬T0⇒T2 := ocount
   T3 = T1 & 1;          |           T3 := T1&1
   ocount = T2 + T3;     |           ocount′ := T2 + T3
   idata = T1 >> 1;      |           idata′ := T1 >> 1
   goto L2;              |           L2′
```

Figure 20. From a generic intermediate representation to propositions

On line 1 for instance, we associate the instruction $\mathsf{T1} := \mathsf{idata}$ to the proposition $\mathsf{L2} \Rightarrow \mathsf{T1} := \mathsf{idata}$. The variable L2 is a boolean that is true iff the block of label L2 is being executed. Hence, the proposition says that, if the label L2 is being executed, then T1 is equal to idata. All propositions are conditioned by L2 to mean that they hold when block L2 is executed. The extent of a proposition is the duration of a reaction. A reaction can be an arbitrarily long yet finite period of time provided that every variable or register changes its value at most once during that period. For instance, consider the instruction if T0 then goto L3. It is likely that label L3 will, just as L2, perform some operation on the input idata. Therefore, its execution is delayed until after the current reaction. We refer to $\mathsf{L3}'$ as the next value of the state variable L3, to indicate that it will be active during the next reaction. Hence, the proposition $\mathsf{L2} \Rightarrow \mathsf{T0} \Rightarrow \mathsf{L3}'$ says that control will be passed to L3 at the next reaction when control is presently at L2 and when T0 is true. The instructions that follow this test are conditioned by the negative $\neg\mathsf{T0}$, this means: "in the block L2 and not in its branch to L3".

6.2.3. Behavioral type inference.

In [23], we define behavioral type inference by induction on the formal syntax of programs pgm. To define it, we assume a finite set of program labels L. To each block of label L, the inference system associates a boolean proposition L, called the *input clock*, and L^{exit}, called its *output clock*. The proposition L is true iff the block L is active (the program counter is at L). The proposition L^{exit} is true iff the execution of block L terminates. The relation defined by the behavioral type system has the form:

$$\boxed{e_0, \mathcal{E} \vdash L : blk : \langle P, e_1 \rangle}$$

Figure 21.

where e_0 denotes the input clock of the block of instructions blk, L is its label, P the proposition to denote its behavior, and e_1 its output or continuation clock. The type environment $\mathcal{E}$ gives the behavior of methods and functions defined in the context of the program.

Example 1 Let us zoom on the block L2 of the ones counter (Figure 20, repeated below). On the first line, for instance, we associate the instruction $\mathsf{T1} = \mathsf{idata}$ of block label L2 to the proposition $\mathsf{L2} \Rightarrow \mathsf{T1} = \mathsf{idata}$. In this proposition, the new variable L2 is a boolean that is true iff the label L2 is being executed. So, the proposition says that, if the label L2 is being executed, then T1 is always equal to idata. If not, then another proposition may hold. In our case, all subsequent propositions are conditioned by L2 meaning that they hold when L2 is being executed. Next, consider the instruction `if T0 then L3`. It is likely that label L3 will, just as L2, perform some operation on the input idata. Therefore, we delay its execution until after the current reaction and refer to L3′ as the next value of the state variable L3, to indicate that it will be active during the next reaction. Hence, the proposition $\mathsf{L2} \Rightarrow \mathsf{T0} \Rightarrow \mathsf{L3}'$ says that control passes to L3 at the next reaction when control is presently at L2 and when T0 is true. The instructions that follow this test are conditioned by the negative $\neg$T0, this means: "in the block L2 and not in its branch to L3".

L2:. . . if T0 then goto L3; ⋮ goto L2;	L2⇒ . . .T0 ⇒L3′ ¬T0⇒⋮ L2′

Figure 22.

Type assignement for native and external method invocations $x = f(y^*)$ is taken from an environment $\mathcal{E}$. It is given the name of the actual parameters $x_{1\cdots k}$, of the result x and of the input clock e. $\mathcal{E}(f)(x_{1\cdots k}xe)$ yields the corresponding behavioral type $\langle P, e_1 \rangle$.

Example 2 As an example, the wait-notify protocol used in SystemC of Java to arbiter access to shared data is modeled using a boolean flip-flop variable x. The `notify` method defines the next value of the lock x by the negation of its current value at the input clock e. The `wait` method continues activates iff the value of the lock x has changed at the input clock L: $L \wedge (x \neq x')$. Otherwise, at the clock $L \wedge (x = x')$, the control is passed to L by a delayed transition $e \setminus \hat{y} \Rightarrow L'$.

Consider the wait-notify protocol at blocks L1 and L3, next. The wait instruction continues if L1 receives control and if the value of lock has changed (proposition $\mathsf{lock} \neq \mathsf{lock}'$). If so, the block is executed and the control is passed to the block L2. Otherwise it is passed to the block L1 again.

$$\begin{array}{rcl}\mathcal{E}(\texttt{notify}) & = & \lambda xe.\langle e \Rightarrow (x' = \neg x), e\rangle \\ \mathcal{E}(\texttt{wait}) & = & \lambda xL.\langle L \wedge (x = x') \Rightarrow L', L \wedge (x \neq x')\rangle\end{array}$$

Figure 23.

L1:wait (lock); ⋮ goto L2;	L1 ∧ (lock = lock′)⇒L1′ ⋮ L1 ∧ (lock ≠ lock′)⇒L2′	L3:notify (lock); ⋮ goto L1;	L3⇒lock′ = ¬lock ⋮ L1′

Figure 24.

6.3. Applications of behavioral modeling

Participants: David Berner, Paul Le Guernic, Jean-Pierre Talpin.

6.3.1. Module checking.

In [23], we define a behavioral module checking algorithm based on the type system presented in the previous section. This system allows to give guarantees on the correct assembly of embedded system components based on type-based assumptions on their behavior. As an example, consider a SystemC class m_0 whose virtual fields are the clocks x, y and a procedure f. Assume an explicit behavioral type declaration $badhbox(f, Q)$ which associates f with a description of its behavior: the proposition Q denotes its expected functionality. Let us associate the interface m_0 with the class parameter m_1 of a template class m_2. The interface m_0 now gives a behavioral type to the method f in the class parameter m_1 expected by the module m_2. The assumption Q on the behavior of $m_1.f$ is required to provide a guarantee on the behavior of the module m_2 produced by the template class. Module m_3 is a candidate parameter for m_2. It structurally implements the interface m_0 and is annotated with the guarantee $badhbox(f, P)$, where P is the type of pgm. Now, let m_4 be the class defined by the instantiation of the template m_2 and the parameter m_3. To check the compatibility of the actual parameter m_3 with the formal parameter m_0, we check the containment of the behaviors denoted by the proposition P (the type of the actual parameter) in the proposition Q (the type of the formal parameter). This amounts to check that P implies Q, either by model checking (if Q contains state transitions) or by static checking (if Q is a "stateless" property).

```
class m0 { virtual sc_clock x, y; virtual void f() {} #TYPE(f, Q) };
template ⟨class m1⟩ #TYPE(m1, m0)
  SC_MODULE(m2) { SC_CTOR(m2) { SC_THREAD(m1.f) sensitive ≪ x } };
class m3 { sc_clock x, y; void f() { pgm} #TYPE(f, P) };
m2⟨m3⟩ m4;
```

Figure 25.

6.3.2. Conformance checking.

Just as the multi-clocked synchronous formalism Signal it is based upon, our type system allows for the refinement-based design methodologies considered in [18][17] to be easily implemented. Checking the correct refinement of an initial module, of type P, by its upgrade, of type Q, amounts to checking that the final guarantee Q satisfies the initial assumptions P. In [18], this is implemented by compositionally model checking that Q is finitely flow-equivalent to P. Figure 26 describes a typical case study of conformance checking. We consider the refinement of the C model of an even parity checker from a high-level design abstraction, left, where communication is abstracted by shared variables and a lock, to an architecture-level design abstraction, right, where the communication medium is refined by the insertion of a channel implementing a double handshake protocol. Checking conformance of the architecture-level design with respect to its system-level abstraction amounts to checking that both designs are flow equivalent.

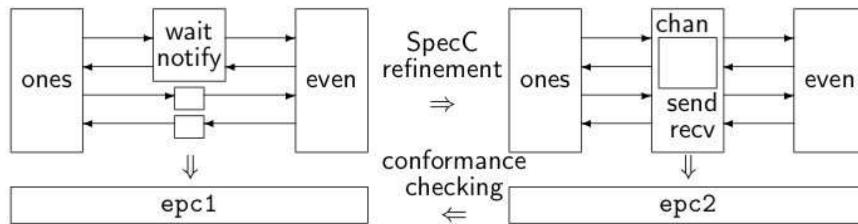

Figure 26. Conformance-checking EPC refinements

6.3.3. Rethreading.

Because our type system entirely models the control and data-flow of application components and architecture functionalities, one can operate global optimization on the whole model of the application. Signal, in particular, implements the STS notation of our type system using data-flow equations and allows for the generation of sequential code by employing a global control-flow graph transformation called hierarchization [1].

Hierarchization consists of hooking elementary control flow graphs (in the form of if-then-else structures). For instance, right, let h3 be a clock computed using h1 and h2. Let h be the head of a tree from which h1 and h2 are computed (an if-then-else), h3 is computed after h1 and h2 and placed under h.

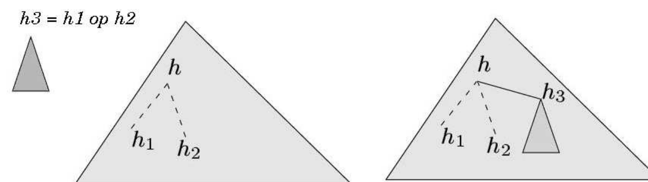

Figure 27.

Operating this transformation on the model of a multi-threaded application results in merging all threads into a single control-flow graph whose scheduler foot-prints sequentially processes each elementary execution block upon a particular condition. In [49], we report a 300% average speedup resulting of applying this optimization to real-time Java programs compared to their execution using a commercial compiler.

6.3.4. Modular design checking by components abstraction.

Reusing intellectual property can help to cope with the growing system complexity. But components often are created for a special purpose, and even if they are sufficiently general, they behave differently in different environments. A lack of pertinent interface descriptions impedes an unambiguous reuse. In many cases subtleties that only the component's original developers are familiar with are indispensable for efficient integration and testing.

We address this problem in [19][15] by providing components with interfaces that not only comprise the components' input / output signals and their types but also causal and synchrony relations between signals. This enables more exhaustive checks for the compatibility of components. Our approach consists in putting the polychronous model of computation to work in the context of system-level design languages such as SystemC. We provide imperative system components with formal behavioral interfaces in different levels of detail. These interfaces can be used to prove formal properties of the components as well as on the composition of components.

The purpose of the case-study [19] is to demonstrate the benefits of modularly associating each SystemC module to a behavioral type interface to perform optimizations and verifications which are modular and yet sensitive to the architecture in which modules or components are placed as reflected by the architecture's behavioral type and by application of an assumption-guarantee reasoning principle. A more recent and larger experiment applies the principles presented in this article to co-modeling by considering predefined SystemC components and connecting them around a bus architecture by giving a synchronous data-flow model of the interconnection wrappers.

First the SystemC code is analyzed in a preprocessing step and some types are replaced for better conversion results. Then a static single assignment intermediate representation is generated. From this representation, clock and scheduling relations are extracted that serve as a basis for the generation of Signal code. The compilation of the signal code performs static checking for types, dependencies, and clock constraints. This results in a highly reliable connection of components as the synchronous composition - once successfully performed - rules out many sources of error that are not checked for in a common type checking system. As the Signal program represents a formal model, also dynamic properties can be checked for, reaching into formal verification with reasonably small additional effort. For the model checker we use, the model has to be abstracted or transformed into a Boolean version.

Properties pertaining on common design errors can easily be expressed and checked using our type system, as case-studied in [19]. Whereas related approaches consist of proposing an ad-hoc type system for analyzing a specific pattern of design errors: race conditions, deadlocks, threads termination; and in a given programming language: Java, C, SystemC, our type system provides a generic framework to perform verification via model checking of behavioral properties of embedded systems described using imperative programming languages.

- *Termination.* A common design error found in embedded system design is the unexpected termination of a thread due to, e.g., an uncaught exception. Here, the termination of a thread f can simply be expressed by the accessibility of the property $f^{exit} = 1$. Unexpected termination can hence be avoided by model-checking the property that $f^{exit} = 0$ is an invariant of f.
- *Deadlocks.* Another common design error is a wait statement that does not match a notification and yields the thread to block. Let $L_{1 \cdots n}$ be the clocks of the blocks $L_{1 \cdots n}$ in which a lock x is notified. Waiting for x at a given label L eventually terminates if P implies the invariant $L \wedge \neg(\wedge_{i=1}^{n} L_i) = 0$.
- *Races.* Similarly, concurrent write accesses to a variable x shared by parallel threads can be checked exclusive by considering the input clocks $e_{1,..n}$ of all write statements $x = f(y, z)$ by verifying that P implies the invariant $(e_i \wedge (\vee_{j \neq i} e_j)) = 0$, $\forall i = 1, ..n$.

6.4. A UML profile for Real-Time and Embedded Systems

Participants: Thierry Gautier, Jean-Pierre Talpin.

For several years, we have studied in several aspects relationships between the UML notation and the synchronous model of SIGNAL. More specifically, we have proposed the BDL notation and defined a translation of UML state diagrams into this notation [52]. Then, in the RNTL ACOTRIS project (see http://www.acotris.c-s.fr), we have defined and experimented models of translation of applications specified using the UML/ACCORD methodology to SIGNAL [50].

The CARROL Research Programme (see http://www.carroll-research.org), which allows specific collaborations between teams from INRIA, CEA and THALES Research and Technology, has been the right vector to further develop the idea of a synchronous UML model, through the PROTES project. This project groups together the ESPRESSO, AOSTE and DART project-teams from INRIA, the CEA-List, and THALES Communications France, THALES Airborne Systems, THALES Underwater Systems. The aim of the project is to define and standardize at the Object Management Group (OMG) a UML 2.0 profile for real-time embedded systems.

Our proposal is based on the models of synchronous programming (with logical time), on models of synchronous hierarchical state diagrams such as SyncCharts, on the representation of data-flow by activity diagrams driven by control clocks, and on the AAA methodology based on SynDEx. The approach advocates distinct modelizations for the structures of the functional application and the deployment architectural platform, which allows architecture exploration for optimization and efficient matching of the mapping of functions onto resources.

The profile will be included in a larger proposal, which, in particular, will take up the SPT profile (Schedulability, Performance and Time) in accordance with UML 2.0, and would take into account SysML (System modeling), AADL (for avionics and automotive) and UML4SoC (System on Chips modeling) initiatives.

Following the OMG procedure, the first step has been to write a *Request for Proposal* (RFP): “UML Profile for Modeling and Analysis of Real-Time and Embedded Systems”, which is now available as an OMG document [24]. It mainly describes the requirements for the profile. It is expected that an official vote on this RFP will take place at the January 2005 OMG meeting.

6.5. Behavioral inheritance in a synchronous model of computation

Participants: Mickael Kerboeuf, Jean-Pierre Talpin.

In [16], we propose a synchronous model of computation that supports the incremental, object-oriented design of synchronous system components. This model consists of a core algebraic formalism, akin to Pnueli’s synchronous transition systems [47], that adapts the paradigm of synchronous programming to the notions of *encapsulation* and of *behavioral inheritance* with *overriding*, borrowed to object-oriented programming. The classical notion of class is introduced as an abstract parameterized and encapsulated process. An object is an instance of a class. An inheritance operator is defined at the class level. It refines the behavior of an initial class with a special class that corresponds to the notion of wrapper. A concurrent behavioral inheritance operator is defined in terms of synchronous composition by introducing a technique of renaming or rewriting.

Reusability of components is achieved by an encapsulation mechanism. A static interpretation of inheritance allows a natural formulation of behavior refinement where overriding is taken into account. The proposed model supports a compile-time resolution of inheritance: overriding is interpreted statically and the inheritance combinator is translated into a primitive synchronous constructs. The benefits of our approach are illustrated by the adaptation of services to a plain old telephone service specification.

6.5.1. Example

Let *balance* be a process that implements a voting balance counter, i.e. it counts the number of times a signal x is true *minus* the number of times it is false. The class C_{balance} is an encapsulation of the process balance. The initial definition is encapsulated within a structure which specifies its *interface*: $[C]$ is the class parameter

which provides the signal x used by the balance. The special parameter self refers to the class C_{balance} itself (which provides the signals n and m). The scope of m is restricted to the definition of the class. m is a *private* signal. We wish to modify the class C_{balance} in order to incorporate a reset signal r (provided by the class

$$C_{\mathsf{balance}} \stackrel{\mathsf{def}}{=} \begin{array}{l}[C]\,.\\ \left[1 + \left((m{=}(\mathsf{pre}\,0)\,\mathsf{self}.n) \;\middle|\; \left(\begin{array}{rr} & (\mathsf{when}\,C.x)\,|\,(n{=}\mathsf{self}.m{+}1)\\ + & (\mathsf{when}\,(\mathsf{not}\,C.x))\,|\,(n{=}\mathsf{self}.m{-}1)\end{array}\right)\right)/m\right]\end{array}$$

Figure 28.

that defines x). The signal r can be invoked only if x is present. It is used to reset the balance to 0. The state of C_{balance} is managed by a *private* signal m. Implementing this upgrade without inheritance would break encapsulation. Using inheritance, it amounts to adding a *wrapper* (on the right hand-side) to the initial class as follows:

$$\begin{array}{l} C_{\mathsf{resettable_balance}} \stackrel{\mathsf{def}}{=}\\ \quad [C]\,.\\ \quad \left[1 + \left((m{=}(\mathsf{pre}\,0)\,\mathsf{self}.n) \;\middle|\; \left(\begin{array}{rr} & (\mathsf{when}\,C.x)\,|\,(n{=}\mathsf{self}.m{+}1)\\ + & (\mathsf{when}\,(\mathsf{not}\,C.x))\,|\,(n{=}\mathsf{self}.m{-}1)\end{array}\right)\right)/m\right]\\ \quad \&\\ \quad [C']\,.\,[\,]\,.\\ \quad \left[1 + (n{=}\mathsf{super}.n) + \left(\begin{array}{ll} & (\mathsf{event}\;\mathsf{super}.n)\\ | & (\mathsf{event}\;C'.r)\\ | & (n{=}0)\end{array}\right)\right]\end{array}$$

Figure 29.

As its syntax suggests, inheritance & is basically a sort of oriented (and non commutative) synchronous composition. There are two parts in the interface of a wrapper. The first part ($[C']$ in the example) enables to *reuse* the parameters of the initial class ($[C]$ of C_{balance}). The second part possibly introduces new parameters ([] in the example: no new parameters).

Stuttering is still enabled by the wrapper (reaction 1). The second reaction $(n = \mathsf{super}.n)$ is enabled if and only if r is absent: it can be triggered in a context that provides values for (and only for) n and $\mathsf{super}.n$. In particular, the presence of r (referenced in the third reaction of the process) inhibits the reaction $(n = \mathsf{super}.n)$. Hence, if r is absent, the second reaction is triggered and the previous version of n prevails. The third reaction specifies that n provides 0 instead of $\mathsf{super}.n$ when r, provided the class by C' (alias C) is present. We introduce a renaming scheme to merge C_{balance} and its wrapper into a unique base class. $C_{\mathsf{resettable_balance}}$ is then equivalent to the following class:

Renaming in the initial class and the wrapper is *selective*. The signal n is overridden. In order to enable the *coexistence* of its initial definition and its new one within a same structure, n is changed into a new name $\mathbf{n}'$

$$C_{\mathsf{resettable_balance}} \equiv$$
$$[C] \cdot \left[\left(\begin{array}{l} 1 + \left((m{=}(\mathsf{pre}\ 0)\ \mathsf{self}.n) \ \middle|\ \left(\begin{array}{rl} & (\mathsf{when}\ C.x) \mid (\mathbf{n}'{=}\mathsf{self}.m{+}1) \\ + & (\mathsf{when}\ (\mathsf{not}\ C.x)) \mid (\mathbf{n}'{=}\mathsf{self}.m{-}1) \end{array} \right) \right) / m \\ \mid \\ 1 + (n{=}\mathsf{self}.\mathbf{n}') + \left(\begin{array}{ll} & (\mathsf{event}\ \mathsf{self}.\mathbf{n}') \\ \mid & (\mathsf{event}\ C.r) \\ \mid & (n{=}0) \end{array} \right) \end{array} \right) / \mathbf{n}' \right]$$

Figure 30.

(whose scope is restricted), *only where it is initially defined*. Where n is only *used* (in the definition of m), n remains unchanged. *Thus, the signal m still maintains a delayed version of (the new version of) n.*

The reactions added by the wrapper are also adapted. The previous references to the "super" signal n are changed into references to the "self" signal $\mathbf{n}'$. Finally, the parameters C and C' are unified in the global structure under the name C. The signals x and r are now provided by the same class C. Notice that it is not possible to extend the initial behavior just by adding a reaction outside the scope of m, because when n is reset to 0, m must register it. It is therefore not possible to achieve the modification without inheritance, or without breaking the encapsulation.

6.6. Modeling and validation of asynchronous systems in synchronous frameworks

Participants: Paul Le Guernic, Jean-Pierre Talpin.

Synchronous languages have been extensively used in the (co-)design of software and hardware systems [3]. Applying synchronous languages to real-world designs revealed their strong and weak points over time. Abstract and easy to learn and use syntax, formal and succinct semantics (which paved the way for efficient simulation and verification tools) are among the strong points of such languages. However, the synchronous assumption turns out to be a limiting factor. On one side of the spectrum, in distributed real-time systems, providing a single, fully synchronized clock over distributed nodes may be very expensive or actually infeasible. On the other side of the spectrum, in nano-scale system design, the propagation delay of the clock over the chip size becomes a major obstacle in providing a single synchronized clock. Thus, all these domains call for a mix of synchronous and asynchronous design patterns. *Globally Asynchronous, Locally Synchronous (GALS)* designs have emerged in the recent years in response to the above mentioned challenges and have received major attention from the system level design community. In GALS design, the system is composed of synchronous components that have their local synchronous clock structures and communicate using asynchronous schemes. There have been several attempts to formalize GALS design.

In [21] we address the problem of modeling and validating asynchronous composition of synchronous components in the multi-clock synchronous programming framework Signal. The main goal of such an approach is to leverage the simulation and model checking toolkit existing for frameworks like Polychrony. Our solution can be seen as a formal methodology for composing existing IP blocks, designed with synchronous assumptions, in an asynchronous fashion to satisfy the demands of tomorrow's GALS designs.

Our approach can be summarized as developing a design consisting of asynchronously composed components within a synchronous framework. Since true asynchrony does not exist in synchronous design frameworks, we seek for a desynchronizing protocol to match the asynchronous model. Finding such a protocol brings about the possibility of formally investigating the behavior of synchronous components in asynchronous environment.

We establish the theoretical model of asynchronous composition in Signal and its implementation using FIFO buffers. In addition to that, we propose a practical design template to estimate the buffer size. The proposed approach allows for efficient analysis and implementation of asynchronous designs. Furthermore, it brings about the possibility of specifying GALS systems in the synchronous framework and benefitting from the tooling around it.

Studying compositionality and stability issues in the buffer size proof and estimation remains as one of our future research topics. We are also looking at constructive algorithms based on the clock dependency graph to make the buffer size estimation and proof automatic. Using a program morphism approach is another possibility for simulating the program behavior and estimating the buffer size.

6.7. A compositional design methodology for refinement checking

Participants: Paul Le Guernic, Jean-Pierre Talpin.

The polychronous model of computation [8] accurately renders the synchronous hypothesis implemented in the multi-clocked data-flow notation Signal. In an embedded architecture, however, the flow of a signal usually slides from another as the result of finite delays incurred by resource-bounded protocols, e.g. `fifo` buffers. In [18], we formulate the properties implied by this practice to establish a compositional correctness criterion for a concrete design refinement methodology.

We start from the model of a one-place `fifo` buffer in Signal, Figure 31, which we will use to draw the spectrum of possible timing relations considered, modelled and checked in the context of the present case study. The processing of process `fifo` is decomposed into two functionalities. One is the process `access` which defines the necessary timing constraints on the input signal `i` and output signal `o` via the delayed value of the boolean signal `b`: `fifo` can accept an input at the next time sample iff `b` is true. The other functionality of `fifo` is the process `current` of Figure 14.

```
process access = (? i, o ! )
  (| a := ^o default (not ^i) default b
   | b := a$1 init false
   | i ^= when b
   | o ^= when not b
   |) / a, b
```

```
process current = (? i ! o)
  (| o:= (i cell ^o init false) when ^o
   |)
process fifo = (? i ! o)
  (| access(i, o) | o := current (i)
   |)
```

Figure 31. A one-place first-in first-out buffer in Signal

In [18] we formalize the relation implied by the process `fifo` in the polychronous model of computation under the notion of finite relaxation. Finite relaxation yields the notion of finite flow equivalence $\approx^*$ to account for the equivalence of two processes modulo bounded `fifo`s. Then, we say that a process P is finitely flow-preserving iff given flow-equivalent inputs, it can only produce behaviors that are finitely flow equivalent. Examples of finitely flow-preserving processes are endochronous processes. An endochronous process receives flow equivalent inputs and produces clock-equivalent outputs. In turn, finite-flow equivalence relates to the notion of weak-endochrony independently proposed by Potop et al. [48]. A refinement-based design methodology based on the property of finite flow-equivalence consists of characterizing sufficient invariants for a given model transformation to preserve flows:

The refinement of p by q is *finitely flow-invariant* iff, for all behaviors b and c of p and q, if inputs along b and c are asynchronously flow equivalent then outputs along b and c are finitely flow equivalent.

The property of finite flow-invariance is a very general methodological criterion which can be applied to checking the correctness of model transformations such as protocol synthesis or desynchronization. It provides all necessary elements to define an observer giving sufficient conditions for equivalence to hold and be provable by model checking. For instance, consider the template Signal process `observer` parameterized by the notation `{P, Q}` over two processes named `P` and `Q` which we want to check equivalent.

```
process observer = {P, Q} (? i ! o)
    (| o := fifo (P (buffer (i))) = fifo (Q (buffer (i))) |);
```

Figure 32.

The `observer` receives an input signal `i`. This input signal is used to generate two desynchronized input signals with `buffer`. Inputs are injected to P and Q and outputs collected with `fifo`. If the output of the `observer` is always true then `P` and `Q` are finitely flow-equivalent.

6.8. Polychronous Modeling and Evaluation of Real-Time Systems

Participants: Loïc Besnard, Abdoulaye Gamatié, Paul Le Guernic, Thierry Gautier.

Real-time systems are often critical because of the high human and economic stakes involved in domains where such systems are encountered (e.g. nuclear power plants, avionics). Therefore, their development requires highly reliable methods. We propose a methodology for the design of real-time systems based on the polychronous model [14]. This methodology allows to design systems that contain asynchronous mechanisms using the synchronous approach.

First, we investigate how to characterize real-time behaviors in the polychronous model. For that, we adapt the semantic model [8] in order to be able to cope with a few real-time issues. These include the following aspects: *temporal scalability* (i.e., how to describe the deployment of a given component on different execution platforms), *modeling of interrupts within real-time executions*, *types of real-time constraints* imposed to a system (we distinguish the constraints involved by an environment from those involved by the execution platform), *temporal refinement* (based on the over-sampling mechanism of the Signal language).

Then, we address verification and analysis issues of polychronous specifications. We mainly identify two ways: *i) by applying to a given program syntactic transformations* that yield some abstraction, which reflects the properties of interest in the program; or *ii) by considering program observers* (the observer of a program `P` is another program, which shares some variables with `P` and does not constrain the behavior of `P`).

We also advocate a general methodology for the design of distributed real-time systems within Polychrony. In particular, avionics has been considered as privileged application domain. We are especially interested in the design of applications following the *Integrated Modular Avionics* (IMA) architecture model that is based on the avionic standard APEX-ARINC [28][29]. This leads to the implementation of a library of components in Signal containing real-time executive services defined by the APEX-ARINC standard.

Finally, the methodology has been experimented by considering a real world case study from the avionic industry [20]. It consists of an application, which cyclically calculates and emits some general parameters and information corresponding to the maintenance phase. It is represented as a *partition* that acquires and treats maintenance messages received from the other subparts of the system during the execution. The partition also interacts with an external device in order to allow an operator to visualize the maintenance information stored in the memory. Finally, it periodically checks the availability of maintenance data, and emits a report message.

A Signal model of the application has been first defined based on our APEX-ARINC models. Then, we addressed the temporal analysis of the application using a technique initially introduced in [43], which consists of using an observer program to check properties of another program (representing the application). Roughly speaking, a Signal program that models an application is recursively composed of sub-programs, where elementary sub-programs belong to the language kernel and are called *atomic nodes*. A profiling of such a process substitutes each signal x with a new signal representing availability dates $date_x$ and automatically replaces atomic nodes with their timing model counter-part ("timing" morphism). The resulting time model (i.e. the observer) is composed with the original functional description of the application, and for each signal x, a synchronization with the signal $date_x$ is added. The resulting program is close to (or even represents exactly) the model of the temporal behavior of the application running on its actual architecture. One can obviously design less strict modeling to get faster simulation (or formal verification); it is sufficient to consider more abstract representations either of the architecture or of the program.

We observe that modularity and abstraction play a central role for scalability in our design approach. Basically, the description of a large application can be achieved with respect to a precise design methodology that consists of specifying first, either completely or partially (by using abstractions), sub-parts of the application. After that, the resulting programs can be composed in order to obtain new components. These components can be also composed and so on, until the application description is complete. The construction of the global simulation model of the application for temporal issues relies on this principle.

Among promising perspectives on this work, we mention further collaborations with Avionics Industry (in particular, with AIRBUS France, one of our partners in the european project IST SAFEAIR that partially supported the present work) for more expriments in order to improve the proposed methodology. On the other hand, the study we began on the characterization of real-time behaviors within the polychronous semantic model raises many challenging issues, which could help to deal with several pragmatic aspects of real-time implementations.

6.9. New features in Polychrony

Participants: Loïc Besnard, Thierry Gautier.

Although we had rough array manipulations in the very first version of Signal, they had not been integrated so far in the Polychrony implementation of Signal V4. Moreover, in collaboration with TNI, we had defined powerful high-level data-flow mechanisms for array restructuring. In response to the request of some users (for example, for image processing applications like the one developed by MBDA during the RNTL Acotris project—see http://www.acotris.c-s.fr), we have defined more user-friendly versions of our operators, syntactically quite close to classical array iterations (this has to be compared, in a different way, to array iterators newly introduced in Lustre V6). We have implemented them and we have provided some illustrative simple examples in the new release of the Signal reference manual [25], available on the Polychrony site. Some work has still to be done, especially for the translation of these structures into the repetitive constructs of SynDEx, thus allowing a full application of the associated AAA methodology (http://www-rocq.inria.fr/syndex).

The notion of *assumption* has been implemented this year. It is possible to specify that a boolean must have the value "true" when it is present. This mechanism can in particular specify hypotheses on some inputs of the model. A property specified by an assertion can be assumed by the clock calculus. In Signal, the equation `C ^= when C` specifies a constraint which is added to the system: `C` is a boolean which is always "true" when it is present. It must be verified (proved). If it is proved, the constraint disappears, otherwise it is kept. With the adding of the assumption `assert C`, the constraint is assumed to be satisfied and `C` and `when C` are unified. However, it is possible to generate code for assertions to verify that they are satisfied at run-time.

Concerning the integrated development environment Polychrony 5.1, a Signal mode for Emacs has been provided to the users. Besides existing Linux and SunOS versions, Polychrony has been ported to MacOS. The next step will be "Windows XP world": a version, using Cygwin, has been build but is not yet provided to the users. Our objective is to provide a (pure) Windows XP version for the end of 2004.

A mid-term goal is now to provide an *open-source* diffusion of the environment. To achieve this, we are currently using the documentation system Doxygen (http://www.stack.nl/~dimitri/doxygen/) to documente the sources of Polychrony. This work is performed in parallel with the packaging of the data structures of Polychrony as libraries. The next step will be the definition of the user interface of these libraries. The aim is the use of these APIs for the definition of model transformations described in high-level IDE tools (UML-based, GME “Model-Integrated Computing”...).

Another use of the APIs is the importation of models (SystemC, real-time Java...) in Polychrony. In particular, we study the transformation of Gnu SSA to Polychrony. *Static Single Assignment* (SSA) is a intermediate representation of the GCC compiler.

7. Contracts and Grants with Industry

7.1. Carroll project Protes (10/2003-10/2005)

Participants: Thierry Gautier, Jean-Pierre Talpin.

The partners of the **CARROLL project Protes** (http://www.carroll-research.org) are Thales, CEA-List and the INRIA project-teams Espresso, Aoste and Dart. The aim of the project Protes is to propose a UML profile for real-time and embedded systems and to defend it before the OMG. The participation of the project-team to this collaboration is addressed section 6.4.

7.2. Network of excellence Artist

Participants: Thierry Gautier, Paul Le Guernic, Jean-Pierre Talpin.

The Espresso project-team is involved in the activity of the Artist network of excellence (http://www.artist-embedded.org). The activity of this network consists of the definition of a road-map on embedded system design.

7.3. Network of excellence Artist2

Participants: Thierry Gautier, Paul Le Guernic, Jean-Pierre Talpin.

The Espresso project-team is involved in the activity of the Artist2 network of excellence. The URL http://www.artist-embedded.org/FP6) gives a detailed description of the scope and aims of the network.

8. Other Grants and Activities

8.1. INRIA associated projects program

Participants: David Berner, Paul Le Guernic, Jean-Pierre Talpin.

In the frame of the renewed cooperation of the Espresso project-team with Virginia Tech and UC San Diego through the associated project program of INRIA, the following milestones have been accomplished:

- The participants edited a book on formal methods and models for system design [12] published by Kluwer Academic Publishers including a selection of the best articles presented during the previous edition of MEMOCODE at Mont Saint-Michel in 2003.

- The Second ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'2004 [13]) took place at the Hilton Harbor Island in San Diego on June 22nd to 25th, 2004. The conference started on day one with a keynotes speech of Pr. Randy Bryant on "System Modeling and Verification with UCLID". This was followed by presentations throughout the day on the topics of modeling languages, model checking and synthesis and finished with a tutorial on "Efficient, correct RTL from high-level specifications" given by Rishiyur Nikhil, CTO of Bluespec Incorporated.
  Day two of the conference began with the keynotes speech of David Dill from Stanford University on "The battle of accountable voting systems" and was followed by presentations on formal verification. The afternoon was devoted to a successful industry panel chaired by Tevfik Bultan of UC Santa Barbara, and whose participants included research officials from Mentor Graphics, Cadence, Microsoft, Intel, Synopsis and NASA's Jet Propulsion Laboratory. One underlying theme was the convergence of formal methods and industrial trends. To underscore the topic, the panel of industry experts talked about the future of software verification, given that hardware verification has been such an uphill battle. An excellent online streaming video coverage of this panel was provided by Cal-(IT)^2 and is available for on-demand viewing at http://www.calit2.net/research/labs/features/7-29-04_memocode.html
  The conference concluded on Day three with a keynote speech of Pr. Edward Lee from UC Berkeley on "Actor-oriented design" followed by presentations on simulation, testing and compositional verification. There were 18 papers accepted for MEMOCODE'2004, which made up the core content for the presentations and discussions on the topic areas discussed above, giving the conference a continued reputation of high-quality competitive selection. Acknowledgment of support from the Industry and Academia including the UC Discovery and other cooperating entities: the Office of Naval Research, the National Science Foundation, BlueSpec Incorporated and INRIA, was included in all printed material. Complete information on the MEMOCODE conference series can be found on the web at http://memocode.irisa.fr.
- David Berner visited Virginia Tech (October-December 2004) and participated to several articles on XFM [22][15]. He is working with Pr. Sandeep Shukla in the Fermat (Formal Engineering Research using Methods, Abstraction, and Transformations) group of the ECE (Electrical and Computer Engineering) department. The main goal of this exchange is to advance in the definition of behavioral interface descriptions for SystemC program blocks. The interface of a SystemC block can be described as a Signal process, containing signal dependency and clock synchronization information. This information (commonly not used in, nor available to a SystemC compiler), will form the type of this component. If we have sound type information for the components available, type checking will expose problems in the model that would otherwise have been detected much later in the design process. Also types can be the basis of the automatic generation and insertion of interfaces, which would significantly facilitate the development process. Finally the implementation of a component can be formally checked against its type with the help of existing model checkers such as SPIN. Our common topic of investigation, the typing of SystemC components, turns out to be a key capability for the speedup, cost reduction, and formalization for the design process of embedded systems.
- Frederic Doucet (UCSD) and Sandeep Shukla (Virginia Tech) visited project Espresso in July 2004 participating to the research topic of behavioral types inference presented in this report [21][18][23]. Frederic Doucet implemented a promising case study on using Polychrony as a pivot model applied to verification and optimization issues in the design of a mobile telephony platform in SystemC.

9. Dissemination

9.1. Advisory

- Paul Le Guernic is executive board member of the Réseau National en Technologies Logicielles and steering committee member of the Réseau National en Micro-Nano Technologies.
- Paul Le Guernic and Jean-Pierre Talpin are steering committee members of the ACM-IEEE conference on methods and models for codesign (MEMOCODE).
- Jean-Pierre Talpin is external advisory board member of the center of embedded systems at Virginia Tech.
- Jean-Pierre Talpin is organization committee member of the GALS workshop series.

9.2. Conferences

- Jean-Pierre Talpin served as technical program co-chair of ACM-IEEE MEMOCODE'2004.
- Paul Le Guernic served as technical program committee member of the ACM-IEEE MEMOCODE'2004 conference.
- Jean-Pierre Talpin served as technical program committee member for the IEEE DATE'2004 conference.
- Jean-Pierre Talpin served as technical program committee member for the embedded system track of the ACM SAC'2004 symposium.
- Thierry Gautier served as technical program committee member of SLAP'2004 (Synchronous Languages, Applications, and Programming), an ETAPS'2004 workshop.
- Jean-Pierre Talpin served as technical program committee member of FESCA'2004, an ETAPS'2004 workshop.

9.3. Events

- Jean-Pierre Talpin gave a tutorial on "Polychrony for system design" at the ACM-IEEE MEMOCODE conference in June 2004.
- Loïc Besnard and Jean-Piere Talpin gave a demonstration of Polychrony at the EuroMicro Conference http://euromicro2004.ifsic.univ-rennes1.fr.
- Loïc Besnard participated to the Inria-Industry meeting "L'ingénierie du logiciel", January 2004 http://www.inria.fr/valorisation/rencontres.

9.4. Thesis

- Jean-Pierre Talpin served as rapporteur in the thesis defense committee of Christian Brunette at INRIA Sophia-Antipolis, October 20.
- Jean-Pierre Talpin served as examiner in the thesis defense committee of Katell Morin-Allory at IRISA, October 27.

9.5. Teaching

- Thierry Gautier, Bernard Houssais and Loïc Besnard taught on real-time programming at the DESS ISA and DIIC 2 of University of Rennes I.
- Abdoulaye Gamatié gave courses as teaching assistant at the University of Rennes I.
- Mickaël Kerbœuf gave courses as teaching assistant at INSA-Rennes.

9.6. Visits

- Jean-Pierre Talpin visited UC San Diego in August 2004 in the frame of the associated projects program.
- David Berner visited Virginia Tech from Sept. 1st. until Dec. 15th. in the frame of the associated projects program.
- Pr. Sandeep Kumar Shukla (Virginia Tech) visited the project-team as Invited Professor of the University of Rennes in July 2004.
- Mohammad Reza Mousavi (Eindhoven University) visited the project-team in July 2004.
- Frederic Doucet (Virginia Tech) visited the project-team July 2004 in the frame of the associated projects program.
- Carlos Martinez (University of Montevideo) visited the project in March 2004 in the frame of a bilateral cooperation with Uruguay. Carlos Martinez designed and implemented a layered UDP/TCP-IP protocol stack model in Signal that is the subject of forthcomming publication.

10. Bibliography

Major publications by the team in recent years

[1] T. P. AMAGBEGNON, L. BESNARD, P. LE GUERNIC. *Implementation of the Data-flow Synchronous Language Signal*, in "Proceedings of the ACM Symposium on Programming Languages Design and Implementation (PLDI'95)", ACM, 1995, p. 163–173.

[2] A. BENVENISTE, B. CAILLAUD, P. LE GUERNIC. *From synchrony to asynchrony*, in "CONCUR'99, Concurrency Theory, 10th International Conference", J. C. M. BAETEN, S. MAUW (editors)., Lecture Notes in Computer Science, vol. 1664, Springer, August 1999, p. 162–177.

[3] A. BENVENISTE, P. CASPI, S. EDWARDS, N. HALBWACHS, P. LE GUERNIC, R. DE SIMONE. *The Synchronous Languages Twelve Years Later*, in "Proceedings of the IEEE Special issue on Modeling and Design of Embedded Systems", vol. 91(1), 2003.

[4] A. BENVENISTE, P. LE GUERNIC, C. JACQUEMOT. *Synchronous programming with events and relations: the Signal language and its semantics*, in "Science of Computer Programming", vol. 16, 1991, p. 103-149.

[5] A. GAMATIÉ, T. GAUTIER. *Synchronous modeling of avionics applications using the SIGNAL language*, in "9th IEEE Real-time/Embedded technology and Applications symposium", IEEE Press, May 2003.

[6] T. GAUTIER, P. LE GUERNIC. *Code generation in the SACRES project*, in "Towards System Safety, Proceedings of the Safety-critical Systems Symposium, SSS'99, Huntingdon, UK", F. REDMILL, T. ANDERSON (editors)., Springer, February 1999, p. 127–149.

[7] A. KOUNTOURIS, C. WOLINSKI. *High-level Pre-synthesis Optimization Steps using Hierarchical Conditional Dependency Graphs*, in "Proceedings of the EUROMICRO'99, Milan, Italie", IEEE Computer Society Press, August 1999.

[8] P. LE GUERNIC, J.-P. TALPIN, J.-C. LE LANN. *Polychrony for system design*, in "Journal of Circuits, Systems and Computers, Special Issue on Application Specific Hardware Design", 2003.

[9] P. LE GUERNIC, T. GAUTIER. *Data-Flow to von Neumann: the Signal approach*, in "Advanced Topics in Data-Flow Computing", J. L. GAUDIOT, L. BIC (editors)., 1991, p. 413–438.

[10] P. LE GUERNIC, T. GAUTIER, M. LE BORGNE, C. LE MAIRE. *Programming Real-Time Applications with Signal*, in "Proceedings of the IEEE", vol. 79, n° 9, Septembre 1991, p. 1321–1336.

[11] H. MARCHAND, P. BOURNAI, M. LE BORGNE, P. LE GUERNIC. *Synthesis of Discrete-Event Controllers based on the Signal Environment*, in "Discrete Event Dynamic System: Theory and Applications", vol. 10, n° 4, October 2000, p. 347–368.

Books and Monographs

[12] R. GUPTA, P. LE GUERNIC, S. SHUKLA, J.-P. TALPIN (editors). *Formal Methods and Models for System Design*, Kluwer Academic Publishers, 2004.

[13] C. HEITMEYERE, J.-P. TALPIN (editors). *Proceedings of the 2nd. ACM-IEEE conference on methods and models for codesign*, IEEE Press, 2004, http://www.irisa.fr/MEMOCODE.

Doctoral dissertations and Habilitation theses

[14] A. GAMATIÉ. *Modélisation polychrone et évaluation de systèmes temps réel*, PhD Thesis, University of Rennes 1, Ph. D. Thesis, May 2004.

Articles in referred journals and book chapters

[15] D. BERNER, S. SUHAIB, S. SHUKLA, J.-P. TALPIN. *Capturing formal specifications into abstract models, chapter in Formal Methods and Models for System Design*, Kluwer Academic Publishers, 2004.

[16] M. KERBOEUF, J.-P. TALPIN. *Encapsulation and behavioural inheritance in a synchronous model of computation for embedded system services adaptation*, in "Journal of Logics and Algebraic Programming", 2004.

[17] J.-P. TALPIN, D. BERNER, S. SHUKLA, P. LE GUERNIC, A. GAMATIÉ, R. GUPTA. *Behavioral type inference for compositional system design, chapter in Formal Methods and Models for System Design*, Kluwer Academic Publishers, 2004.

[18] J.-P. TALPIN, P. LE GUERNIC, S. K. SHUKLA, R. GUPTA, F. DOUCET. *Formal refinement checking in a system-level design methodology*, in "Fundamenta Informaticae", 2004.

Publications in Conferences and Workshops

[19] D. BERNER, J.-P. TALPIN, P. LE GUERNIC, S. K. SHUKLA. *Modular design through component abstraction*, in "International conference on compilers, architectures and synthesis for embedded systems", ACM Press, 2004.

[20] A. GAMATIÉ, T. GAUTIER, P. LE GUERNIC. *An Example of Synchronous Design of Embedded Real-Time Systems based on IMA*, in "Proceedings of the 10th International Conference on Real-time and Embedded Computing Systems and Applications (RTCSA'2004), Gothenburg, Sweden", August 2004.

[21] M. MOUSAVI, P. LE GUERNIC, J.-P. TALPIN, S. SHUKLA, T. BASTEN. *Modeling and validation of asynchronous systems in synchronous frameworks*, in "Design Analysis and Test Europe", IEEE Press, 2004.

[22] S. SUHAIB, D. MATHAIKUTTY, D. BERNER, S. SHUKLA. *Extreme Formal Modeling (XFM) for Hardware Models*, in "5th International Workshop on Microprocessor Test and Verification", 2004.

[23] J.-P. TALPIN, D. BERNER, S. K. SHUKLA, A. GAMATIÉ, P. LE GUERNIC, R. GUPTA. *A behavioral type inference system for compositional system-on-chip design*, in "Application of Concurrency to System Design", IEEE Press, 2004.

Miscellaneous

[24] *UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (RT/E-ML*, Request For Proposal, OMG Document, October 2004.

[25] L. BESNARD, T. GAUTIER, P. LE GUERNIC. *SIGNAL V4–INRIA version: Reference Manual*, March 2004, http://www.irisa.fr/espresso/Polychrony.

Bibliography in notes

[26] *Design Guidance for Integrated Modular Avionics*, Technical report, n° ARINC Specification 651-1, Airlines Electronic Engineering Committee, 1997.

[27] *Avionics Application Software Standard Interface*, Technical report, n° ARINC Specification 653, Airlines Electronic Engineering Committee, 1997.

[28] AIRLINES ELECTRONIC ENGINEERING COMMITTEE. *ARINC Report 651-1: Design Guidance for Integrated Modular Avionics.*, in "Aeronautical radio, Inc., Annapolis, Maryland", November 1997.

[29] AIRLINES ELECTRONIC ENGINEERING COMMITTEE. *ARINC Specification 653: Avionics Application Software Standard Interface.*, in "Aeronautical radio, Inc., Annapolis, Maryland", January 1997.

[30] T. BALL, B. COOK, S. DAS, S. RAJAMANI. *Refining Approximations in Software Predicate Abstraction*, in "Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, v.

2988", Springer, 2004.

[31] A. BENVENISTE, P. CASPI, L. CARLONI, A. SANGIOVANNI-VINCENTELLI. *Heterogeneous Reactive Systems Modeling and Correct-by-Construction Deployment*, in "Embedded Software Conference (EMSOFT'03)", Springer Verlag, 2003.

[32] D. BEYER, A. CHLIPALA, T. HENZINGER. *he Blast query language for software verification*, in "International Static Analysis Symposium, Lecture Notes in Computer Science, v. 3148", Springer, 2004.

[33] S. CHAKI, E. CLARKE, A. GROCE, S. JHA, H. VEITH. *Modular Verification of Software Components in C*, in "In Transactions on Software Engineering, v. 30", IEEE Press, 2004.

[34] H. CHEN, D. DEAN, D. WAGNER. *Model Checking One Million Lines of C Code*, in "Network and Distributed System Security", ISOC, 2004.

[35] A. GAMATIÉ, T. GAUTIER. *Modeling of modular avionics architectures using the synchronous language SIGNAL*, in "Proceedings of the 14th. Euromicro Conference on Real-Time Systems, work-in-progress session", IEEE Press, 2002.

[36] E. GIMÉNEZ. *Un Calcul de Constructions Infinies et son Application à la Vérification des Systémes Communicants*, Ph. D. Thesis, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, December 1996.

[37] P. GODEFROID. *Software Model Checking: The VeriSoft Approach*, in "Technical Memorandum ITD-03-44189G", Bell Labs, 2003.

[38] J. HATCLIFF, M. DWYER. *Using the Bandera Tool Set to Model-check Properties of Concurrent Java Software*, in "Invited tutorial, conference on concurrency theory, Lectures Notes in Computer Science V. 2154", Springer, 2001.

[39] G. HOLZMANN. *The SPIN model-checker*, Addison-Wesley, 2003.

[40] G. HOLZMANN, M. SMITH. *An automated verification method for distributed systems software based on model extraction*, in "IEEE Transactions on Software Engineering, v. 28", IEEE Press, 2002.

[41] M. KERBOEUF, D. NOWAK, J.-P. TALPIN. *Formal proof of a polychronous protocol for loosely time-triggered architectures*, in "Formal Methods and Software Engineering: 5th International Conference on Formal Engineering Methods", Lecture Notes in Computer Science n. 2885, Springer Verlag, 2003.

[42] M. KERBŒUF, D. NOWAK, J.-P. TALPIN. *The steam-boiler problem in SIGNAL-COQ*, in "International Conference on Theorem Proving in Higher-Order Logics", Lecture Notes in Computer Science, Springer Verlag, 2000.

[43] A. KOUNTOURIS, P. LE GUERNIC. *Profiling of SIGNAL programs and its application in the timing evaluation of design implementations*, in "IEE Colloqium on HW-SW Cosynthesis for Reconfigurable Systems", IEE, 1996.

[44] D. KROENING, E. CLARKE, F. LERDA. *A Tool for Checking ANSI-C Programs*, in "Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, v. 2988", Springer, 2004.

[45] D. NOVILLO. *Tree SSA, a new optimization infrastructure for GCC*, in "GCC developers summit", 2003.

[46] D. NOWAK, J.-R. BEAUVAIS, J.-P. TALPIN. *Co-inductive axiomatization of a synchronous language*, in "International Conference on Theorem Proving in Higher-Order Logics", Lecture Notes in Computer Science, Springer Verlag, 1998.

[47] A. PNUELI, O. SHTRICHMAN, M. SIEGEL. *Translation validation: from Signal to C*, in "Correct System Design Recent Insights and Advance", Lecture Notes in Computer Science, vol. 1710, Springer Verlag, 2000.

[48] D. POTOP, B. CAILLAUD, A. BENVENISTE. *Concurrency in synchronous systems*, in "Applications of concurrency to system design", IEEE Press, 2004.

[49] J.-P. TALPIN, A. GAMATIÉ, D. BERNER, B. LE DEZ, P. LE GUERNIC. *Hard real-time implementation of embedded software in JAVA*, in "International Workshop on scientiFic engIneering of Distributed Java applIcations", Lecture Notes in Computer Science, Springer Verlag.

[50] Y. TANGUY, S. GÉRARD, T. GAUTIER. *Modélisation ACCORD/UML Signal*, Technical report, n° DTSI/SLA/03-621, CEA/Saclay Report, Direction de la recherche technologique, List, October 2003.

[51] W. VISSER, K. HAVELUND, G. BRAT, S. PARK, F. LERDA. *Model Checking Programs*, in "Automated Software Engineering Journal, v. 10", 2003.

[52] Y. WANG. *UML et technologie synchrone pour les systèmes réactifs distribués*, Ph. D. Thesis, Thèse de doctorat de l'Université de Rennes 1, 2001.

[53] B. WERNER. *Une Théorie des Constructions Inductives*, Ph. D. Thesis, Université Paris VII, May 1994.