INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team Espresso

Modélisation de systèmes réactifs polychrones

Rennes - Bretagne-Atlantique

THEME COM

Activity Report

2008

Table of contents

1. Team

Research Scientist
Jean-Pierre Talpin [Research Director, INRIA, HdR]
Thierry Gautier [Research Associate, INRIA]
Paul Le Guernic [Research Director, INRIA]

Technical Staff
Loïc Besnard [Research Engineer, CNRS]

PhD Student
Yann Glouche [INRIA]
Yue Ma [INRIA]
Hugo Métivier [University of Rennes]

Post-Doctoral Fellow
Christian Brunette [Expert Engineer, INRIA]
Julio Peralta [Expert Engineer, INRIA]
Julien Ouy [Expert Engineer, INRIA]
Eric Vecchie [Post-Doctorate, INRIA]

Visiting Scientist
Sandeep Shukla [Invited Professor, Virginia Tech]

Administrative Assistant
Lydie Mabil [Secretary, INRIA]

2. Overall Objectives

2.1. Introduction

The Espresso project-team proposes models, methods and tools for computer-aided design of embedded systems. The model considered by the project-team is polychrony [13]. 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 GeenSys supplies its commercial implementation, RT-Builder, used for industrial scale projects by Snecma/Hispano-Suiza and EADS – Airbus Industries (see http://www.geensys.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 [4]. 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

3.1. 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 [13] 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 [27] 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 $t_3 \neg \leq t_1$.

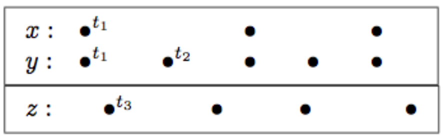


Figure 1. Behavior b over three signals x, y and z in two clock domains

3.1.1.1. Composition

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 $x \in I = \mathrm{vars}(b) \cap \mathrm{vars}(c)$ 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.

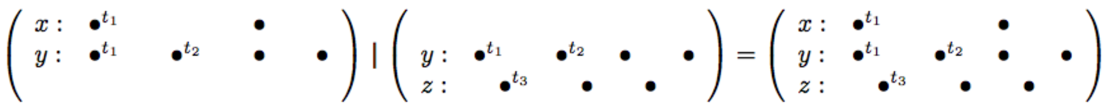


Figure 2. Synchronous composition of $b \in p$ and $c \in q$

3.1.1.2. Scheduling

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 $t' \neg < t$.

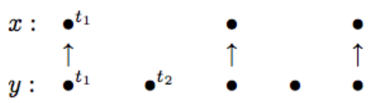


Figure 3. Scheduling relations between simultaneous events

3.1.1.3. Structure

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 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 $d \leq c$.

$$\begin{tikzcd}
x: & \bullet^{t_1} & & \bullet & & \bullet & & x: & \bullet^{t_3} & & \bullet & & \bullet \\
y: & \bullet^{t_1} \arrow[u] & \bullet^{t_2} & \bullet \arrow[u] & \bullet & \bullet \arrow[u] & \leq & y: & \bullet^{t_3} \arrow[u] & \bullet^{t_4} & \bullet \arrow[u] & \bullet & \bullet \arrow[u]
\end{tikzcd}$$

Figure 4. Relating synchronous behaviors by stretching.

3.1.2. A declarative design language

Signal [5] 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 $x = y \,\$\, \texttt{init}\, v$, 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 $x = y \,\$\, \texttt{init}\, v$ 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}.

$$\begin{array}{rcccc}
y & \bullet^{t_1,v_1} & \bullet^{t_2,v_2} & \bullet^{t_3,v_3} & \ldots \\
y \,\$\, \texttt{init}\, v & \bullet^{t_1,v} & \bullet^{t_2,v_1} & \bullet^{t_3,v_2} & \ldots
\end{array}$$

- sampling $x = y\, \texttt{when}\, z$, 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 $x = y\, \texttt{default}\, z$, 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).

$$\begin{tikzcd}
y & & \bullet & \bullet^{t_2,v_2} \arrow[d] & \ldots & y & & \bullet^{t_2,v_2} \arrow[d] & \bullet^{t_3,v_3} \arrow[d] & \ldots \\
y\, \texttt{when}\, z & & & \bullet^{t_2,v_2} & \ldots & y\, \texttt{default}\, z & \bullet^{t_1,v_1} & \bullet^{t_2,v_2} & \bullet^{t_3,v_3} & \ldots \\
z & \bullet & \bullet^{t_1,0} & \bullet^{t_2,1} \arrow[u] & \ldots & z & \bullet^{t_1,v_1} \arrow[u] & & \bullet & \ldots
\end{tikzcd}$$

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)
    |);
```

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
     |);
```

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 specifications

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. $\texttt{when}\,\texttt{not}\, x$) represents the time tags at which a boolean signal x is present and true (resp. false).

The empty clock is written 0 and clock expressions e combined using conjunction, disjunction and symmetric difference. Clock 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'. Explicit scheduling relations $x \rightarrow y\,\texttt{when}\, e$ allow to schedule the calculation of signals (e.g. x after y at the clock e).

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

3.1.3.2. Synchronization and scheduling analysis

A Signal process P corresponds to a system of clock and scheduling relations E that denotes its timing structure. It can be defined by induction on the structure of P using the inference system $P : E$ of Figure 5.

```
x := y$ init v   : ^x ^= ^y
```

```
x := y when z    : ^x ^= ^y when z | y -> x when z
x := y default z : ^x ^= ^y default ^z | y -> x when ^y | z -> x when ^z ^- ^y
```

Figure 5. Clock inference system

3.1.3.3. 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*. A process is said endochronous iff, given a set of input signals and flow-equivalent input behaviors, it has the capability to reconstruct a unique synchronous behavior up to clock-equivalence: the input and output signals 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 hooks elementary control flow graphs (in the form of if-then-else structures) one to the others. Figure 6, 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.

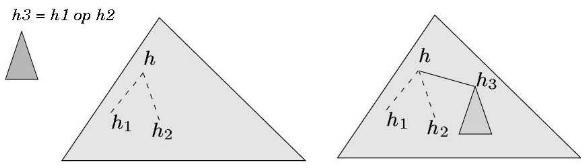


Figure 6. Hierarchization of clocks

4. Application Domains

4.1. 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, Speeds, and through the national ANR projects Topcased, OpenEmbeDD, Spacify. 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.

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, 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.

Polychrony supports the synchronous, multi-clocked, data-flow 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.

Polychrony 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 GeenSys supplies a commercial implementation of Polychrony, called RT-Builder, used for industrial scale projects by Snecma/Hispano-Suiza and Airbus Industries (see http://www.geensys.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. 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).
3. GUIs (in Java and on top of Eclipse) with interactive access to compiling functionalities.

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 (http://www-rocq.inria.fr/syndex).

Polychrony is open-source software under CECILL-B license packaging its data-structure and algorithms as service libraries. These libraries are used by the Eclipse plugins of the SME modeler (Sec. 5.4).

5.2. Integrated Modular Avionics design using Polychrony

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

The Apex interface, defined in the ARINC standard [25], 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 [26]). 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 [6] is now part of the Polychrony distribution 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.

5.3. Multi-clocked mode automata

Keywords: *Generic Modeling Environment*, *Mode automata*, *Model transformation*, *Signal*.

Participants: Jean-Pierre Talpin, Thierry Gautier, Christian Brunette.

Gathering advantages of declarative and imperative approaches, mode automata were originally proposed by Maraninchi et al. [36] to extend the functionality-oriented data-flow paradigm with the capability to model transition systems easily and provide an additional imperative flavor. Similar variants and extensions of the same approach to mix multiple programming paradigms or heterogeneous models of computation [30] have been proposed until recently, the latest advance being the combination of stream functions with automata in [31]. Nowadays, commercial toolsets such as the Esterel Studio's Scade or Matlab/Simulink's Stateflow are largely inspired from similar concepts.

While the introduction of preemption mechanism in the multi-clocked data-flow formalism Signal was previously studied by Rutten et al. in [39], no attempt has been made to extend mode automata with the capability to model multi-clocked systems and multi-rate systems. In [40], we extend Signal-Meta with an inherited metamodel of multi-clocked mode automata. A salient feature is the simplicity incurred by the separation of concerns between data-flow (that expresses structure) and control-flow (that expresses a timing model) that is characteristic to the design methodology of Signal.

While the specification of mode automata in related works requires a primary address on the semantics and on compilation of control, the use of Signal as a foundation allows to waive this specific issue to its analysis and code generation engine Polychrony and clearly exposes the semantics and transformation of mode automata in a much simpler way by making use of clearly separated concerns expressed by guarded commands (data-flow relations) and by clock equations (control-flow relations).

5.4. Eclipse plugins for Polychrony

Keywords: *Eclipse*, *Ecore*, *Meta-modeling*, *Model transformation*, *Signal*.

Participants: Christian Brunette, Loïc Besnard.

We have developed a metamodel and interactive editor of Polychrony in Eclipse. Signal-Meta is the metamodel of the Signal language. It describes all syntactic elements specified in [29]: all Signal operators (e.g. arithmetic, clock synchronization), model (e.g. process frame, module), and construction (e.g. iteration, type declaration).

Signal-Meta has been extended to allow the definition of mode automata, which were originally proposed by Maraninchi et al. [36] to extend the functionality-oriented data-flow paradigm with the capability to model transition systems easily and provide an additional imperative flavor (Sec. 5.3).

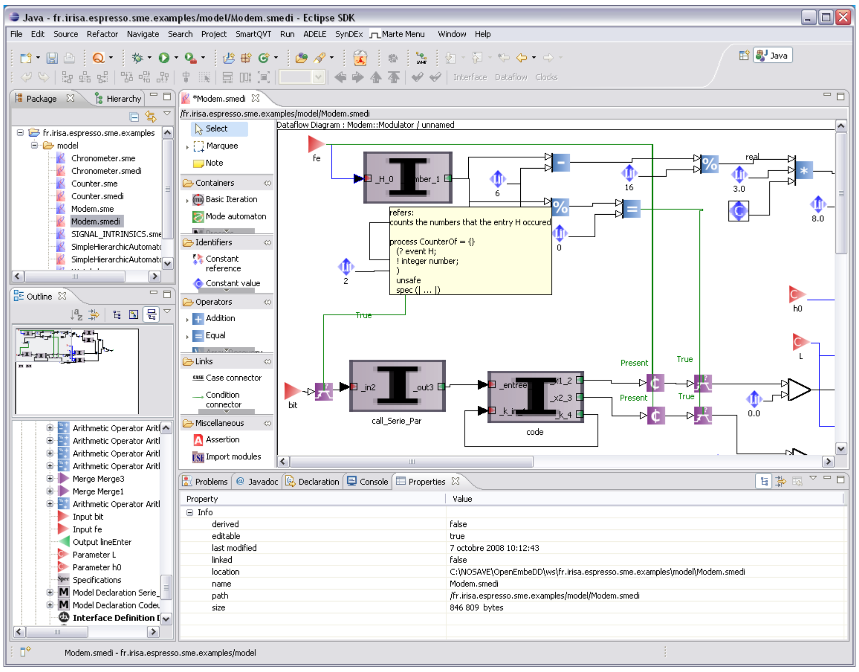
These metamodels aim at providing a user with a graphical framework allowing to model applications using a component-based approach. Application architectures can be easily described by just selecting these components via drag and drop, creating some connections between them and specifying their parameters as component attributes. Using the modeling facilities provided with the Topcased framework, we have created a graphical environment for Polychrony (see figure 7) called SME (Signal-Meta under Eclipse). To hightlight the different parts of the modeling in Signal, we split the modeling of a Signal process in three diagrams: one to model the interface of the process, one to model the computation (or dataflow) part, and one to model all explicit clock relations and dependences.

The SME environment is available through teh Espresso update site [23], in the current OpenEmbeDD distribution [22], or in the TopCased v2.0 experimental distribution [24].

6. New Results

6.1. New features of Polychrony

Participants: Loïc Besnard, Thierry Gautier.


Java - fr.irisa.espresso.sme.examples/model/Modem.smedi - Eclipse SDK
File Edit Source Refactor Navigate Search Project SmartQVT Run ADELE SynDEx Marte Menu Window Help
Package Hierarchy
*Modem.smedi
Dataflow Diagram : Modem::Modulator / unnamed
Select
Marquee
Note
Containers
Basic Iteration
Mode automaton
Identifiers
Constant reference
Constant value
Operators
Addition
Equal
Links
Case connector
Condition connector
Miscellaneous
Assertion
Import modules
Outline
Problems Javadoc Declaration Console Properties

```
refers:
counts the numbers that the entry H occured

process CounterOf = {}
  (? event H;
  ! integer number;
  )
  unsafe
  spec (| ... |)
```

Property	Value
Info	
derived	false
editable	true
last modified	7 octobre 2008 10:12:43
linked	false
location	C:\NOSAVE\OpenEmbeDD\ws\fr.irisa.espresso.sme.examples\model\Modem.smedi
name	Modem.smedi
path	/fr.irisa.espresso.sme.examples/model/Modem.smedi
size	846 809 bytes

Figure 7. Eclipse SME Environment.

Two compiling related features have been particularly studied and implemented this year:

- Integration of distributed code generation in Polychrony.
- Participation to the development of an IDD manipulation library prototype.

6.1.1. Distributed code generation

We have added and tested distributed code generation in Polychrony. The principles are the following:

1. Description of the software dataflow diagram, using the Signal language (program P).
2. Description of the architecture target, using the Signal language.
3. Mapping of the software diagram on the architecture target. So, P is rewritten as $(|P_1|...|P_n|)$ where n is the number of "processors". This step is only a syntactic restructuration of the original program. The Signal process P_i is annotated with a *pragma* "RunOn i". This pragma will be used for partitioning.
4. Global compiling. This phase makes explicit all the synchronizations of the application, then synthesizes, if it exists, the master clock of the program, and detects synchronization constraints. If it is not, the program is made endochronous. The compiling stops if constraints are detected.
5. The partitioning of the resulting graph. At this step, the pragma "RunOn i" is used: all the nodes of the original P_i process are set in the same sub-graph completed with explicited synchronizations. It is important to note that since the program is endochonous, all the sub-graphs are also endochonous, but may have different master clocks. At the end of this step the program is rewritten as $(|PP_1|...|PP_n|)$, in which the synchronization relations between the PP_i's are added in their interface.
6. Adding communication channels. For that purpose, we use, as a possible implementation, the MPI (Message Passing Interface) library [38].

7. Finally, local compiling of the sub-graphs (PP_i). This step follows the code generation scheme with *threads* developed last year. It is based on a partitioning of the graph into *clusters*:
   - Clusters are defined such that each flow of a given cluster depends on the same set of input flows.
   - A cluster can be executed atomically as soon as its inputs are available.
   - There is a specific cluster for state variable updating (called “Cluster delay”).

   This partitioning is used for a multi-thread simulation code, with *dynamical* scheduling:
   - One task per cluster + one task per input/output + one task that manages the iteration steps.
   - The code of clusters is left unchanged, but synchronizations are added.
   - One semaphore Sem_i per task T_i.
   - At the beginning of each iteration step of a task T_i: as many `wait(`Sem_i`)` as T_i has predecessors.
   - At the end of each iteration step of a task T_i: one `signal(`Sem_k`)` for each T_k successor of T_i.
   - The iteration step is managed by a specific task that executes a `signal()` on the semaphores associated with the tasks that have no predecessor, and then, a `wait()` on its own semaphore (a `signal()` on this latter is executed at the end of the task corresponding to the “Cluster delays”).

A few examples have been developed for the distribution of the tool. Now it is currently used to develop a case study for the Spacify project.

6.1.2. IDD manipulation library

In cooperation with Abdoulaye Gamatié (LIFL), we have defined an interval abstraction of Signal programs in order to cope with the limitation of the clock abstraction when dealing with non-logical expressions. The solution we propose consists in considering mixed Boolean-numeric interval formulas instead of propositional formulas for the encoding of the clock algebra. For their efficient manipulation, we have implemented the corresponding data structures, called *interval decision diagrams* IDD$^+$, which are slightly more general than usual IDDs in that their leaves can be any BDD formula [16]. The IDD$^+$ manipulation library has now to be integrated within the Polychrony platform in order to improve the current clock analysis and automatic code generation.

6.2. New features of the Eclipse plug-ins

Keywords: *Eclipse*, *Ecore*, *Meta-modeling*, *Model transformation*.

Participants: Loïc Besnard, Christian Brunette, Jean-Pierre Talpin.

To dynamically check the correctness of a model in Eclipse, the environment is deeply connected to the Polychrony compiler. Through this connection, the facilities of the compiler can be used directly from the Eclipse environment: clock calculus, code generation (C/C++/Java) for simulation, Signal text generation... This connection consists in a Java/C native interface compiled in a native library (for Linux, Windows, and MacOS X/Intel). The principle of the communication (represented in Figure 8) is the following:

- transform the model conform to the SME meta-model to the abstract syntax tree (AST) representation inside the compiler,
- transform this AST representation to an internal graph structure,
- apply the different Polychrony services to this graph (clock resolution, code generation...),
- report the errors provided by the compiler to the graphical part (partially implemented).

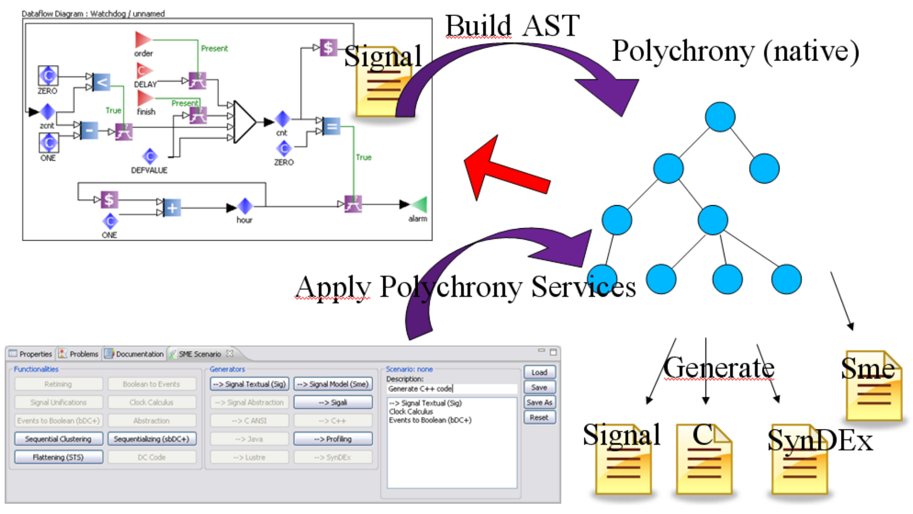


Figure 8. Interaction with Polychrony.

To be able to reuse all existing Signal programs and libraries, Polychrony was also extended to generate SME models from a Signal textual file. The first version of the Signal/SME translator, developed last year, allowed to use only the reflexive editor that visualizes the SME model as a tree. This year, it has been completed to use the graphical view of the editor. Moreover, the generator has been extended for the Signal modules.

6.3. Polychrony integration in the TopCased platform

Participants: Loïc Besnard, Christian Brunette, Julio Peralta.

The Espresso project-team participates to the TopCased (Toolkit in OPen source for Critical Applications and SystEms Development) project, initiative of Airbus. The integration of a tool, such as Polychrony, in TopCased requires to respect the TopCased quality kit. It consists in the production of the several documents that cover the different steps of the development:

- The software development plan which provides the definition of the context and the objectives, the hypotheses and the constraints, the roadmap, the licence, the project organization, the process of the development, the strategy of verification and validation, the document management, the project management.
- The specification of the software, which consists in the definition of the functional requirements, the operational environment requirements (hardware and software environment), the interface requirements (interface with other tools, Topcased...), the performance requirements.
- The design of the software. It is the definition of the architectural design that describes the hierarchy dependency of software components, the design description that defines the interfaces of software components (API).
- The verification plan. It defines the verification means, the description of the tools used to carry out the verifications, the verification environments (environments used for the processing of the verification), the strategy describing all the types of activities done on the tool: integration tests, validation tests, performance tests, regression tests, tests of reused components, analysis, the description of the verifications: for each verification, the objective, the covered specification requirements, the inputs, the actions and expected results, the name of the file for an automatic verification, and the traceability to verify that all the specification requirements are tested.

- The verification result which consists in the definition of the date of the verification campaign and the verified release, the verification results (verification reference, status of the test), and the justifications for the verifications.
- The software configuration plan which provides the list of the objects (files, components, packages, makefiles, installation scripts...) of the tool and the list of the modifications/evolutions of the tool (release notes).
- The installation and administration guide.
- The user guide.

To respect the Topcased Quality Kit, the verification plan and the verification result have still to be completed for Polychrony. Currently, Polychrony is provided in the TopCased distribution as an experimental tool.

6.4. Merging synchrony and asynchrony in a uniform framework

Participants: Paul Le Guernic, Thierry Gautier.

The design of safety-critical embedded systems faces up to strong and heterogeneous constraints that may seem contradictory: continuous model of the environment, partial specifications for abstract properties and functional or non-functional constraints, need for rapid prototyping or simulation, requirement of correctness, safety, most often determinism, multi-tasking implementation, distributed execution... All these constraints have to be managed in a consistent way. A lot of MoCCs (models of computation and communication) have been proposed that are suitable for some design style or execution architecture; most of them handle general parallel asynchronous execution.

We have started a new work to compare the polychronous model with respect to these questions. In this work, we revisit and extend the polychronous model and its associated language Signal to demonstrate its suitability for complementary aspects: to express constraints and abstractions at specification level; to give functional (deterministic) behaviors, i.e., safe executable programs; to execute it following various models of interaction (reactive, pro-active...) on various architectures (centralized, distributed...); to guarantee the correctness (w.r.t. some property) of the different levels of description by the means of correct-by-construction transformations and verifications.

The main idea of the design flow is to start with specifications given as relations (non-deterministic behaviors), to add new constraints such that the program control becomes deterministic and to reinforce execution order (control flow). To distribute this program while preserving its (stream) semantics, i.e., latency insensitivity, we may have to add internal control signals such that the distributed execution is a Kahn process network. A keypoint of this study is to extend Signal with basic asynchronous features.

6.5. Periodic clock relations

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

This work aims at helping a developer to design and to analyse *periodic systems*. Examples of the so-called periodic systems are 4-stroke engines, video converters or some embedded systems in satellites. We define a new operator in Signal to specify constraints on a pair of signals with an ultimately periodic infinite ternary word. This extension of Signal allows to express periodic constraints between pairs of signals. We extend the existing clock calculus of Signal with periodic dependencies.

A first analysis infers the set of periodic relations obtained by composition of the set of constraints specified by the developer.

Embedded systems often need buffers to process delayed transmissions of data. If the developer provides the periods for the production and consumption of data, then the analysis is able to compute the minimal needed size of the FIFO that guarantees correctness. These results are published in [20].

6.6. A contract-based module system

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

Contract-based systems, based on the assume-guarantee paradigm, have become a popular formalism for the modular specification of object-oriented programs. In the context of the development of embedded systems, functional (behavioral) contracts are being applied and become part of mainstream industrial tools.

Our goal is to exploit contracts during the development phase, for instance as a support for early execution (simulation). In this context, we have introduced a new paradigm to express the essence of encapsulation and inheritance in a synchronous and concurrent modeling framework.

A *contract* is a pair (*assumptions, guarantees*). *Assumptions* describe properties expected by a component to be satisfied by the context in which this component is used; on the opposite *guarantees* describe properties that are satisfied by the component when the context satisfies *assumptions*. Such *contract* can be documentary; when a suitable formal model exists *contracts* can be subject to some formal verification tools. We want to provide designers with such a formal model allowing "simple" but powerful and efficient computation on contracts. Thus we define a novel algebraic framework to enable logical reasoning on contracts [20].It is based on two simple concepts.

First, the assumptions and guarantees of a component are defined as filters: assumptions filter the processes a component may accept and guarantees filter the processes a component provides. A filter is the set of processes, whatever their input and output variables are, that are compatible with some properties (or constraints), expressed on the component variables.

Second and foremost, we define a boolean algebra to manipulate filters. This yields an algebraically rich structure which allows us to reason on contracts (to abstract, refine, combine and normalize them). This algebraic model is based on a minimalist model of execution traces, allowing one to adapt it easily to a particular design framework.

We use this algebra to work for the definition of a general purpose module language whose typing paradigm is based on the notion of contract. The type of a module is a contract pertaining assumptions made and guarantees offered by its behaviors. It allows to interface a module with an interface which can be used in varieties of scenarios such as checking the composability of modules or effciently supporting modular compilation.

6.7. Verification of GALS architectures

Participants: Julio Peralta, Loïc Besnard, Jean-Pierre Talpin.

This work is sponsored by the project TOPCASED. It aims to bridge specifications expressed in a subset of the synchronous Signal language with tools for model checking, with the aim of validating such specifications when composed asynchronously (thus forming so-called GALS architectures). In particular, we studied the adequacy of a translation from Signal programs to FIACRE programs. FIACRE, in turn, is a common language for describing synchronous and/or asynchronous models for two model-checker tools: CADP [33] and TINA [28]. However, the synchronous semantics in our source and target languages do not describe communication at the same detail; FIACRE synchronization semantics is too detailed compared with that of Signal programs. Consequently, a small-step operational semantics of Signal seems most appropriate to match with the semantics of FIACRE; moreover, a translation based on a small-step semantics of Signal is what we have been testing on as translation schema with an avionics example. An advantage of our proposed small-step semantics for our subset of SIGNAL programs is that desynchronization is "compositional". A disadvantage is that too much detail in the (FIACRE) translation easily exploses in size, and renders its further exploitation for model-checking untractable. Despite this disadvantage we consider this translation useful since it is correct. Furthermore, we are experimenting on reducing its size by considering the information generated by the Signal compiler, as well as making abstractions on the resulting FIACRE translation through the reduction relations provided in the CADP model-checker, for instance.

As an alternative model-checker we are experimenting with the (Cadence) SMV [37] model checker since the synchronous semantics of models in such language appear closer to that of synchronous Signal programs. More precisely, SMV models seem to reproduce the monochronous semantics of Signal programs, and because Signal specifications are typically polychronous we would like to be able to systematically express "polychronous" SMV models. Alternatively, we envisage using the Signal compiler to generate (whenever possible) Signal programs with a monochronous format since that format may be readily rendered as an SMV specification.

In order to mechanize our experiments, on translation from Signal kernel to FIACRE, we are producing a prototype ATL translator from a subset of the Signal metamodel, SME, into the FIACRE metamodel, under Eclipse. The FIACRE models so produced can be translated into textual form using a tool developed by the VASY team of INRIA-Grenoble. Our experiments with the CADP toolset use a textual interface. The experiments with the SMV model checker is through a textual interface using hand-made translations. Experiments with the SIGALI model checker use a textual interface too.

Our short term goals consider producing an Eclipse plugin enabling automatic translation from a Signal model to a FIACRE model; and, a translation schema from Signal textual form to SMV. Long term goals include the generation of multiple abstractions from source Signal programs or its FIACRE counter-part to enable effective model-checking with CADP and/or TINA; model-checking of polychronous Signal specifications in SMV; and testing of the proposed translations on avionics GALS case-studies.

6.8. Virtual prototyping of avionic architecture descriptions

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

In the context of the TopCased project, we are designing and developing a tool for the virtual prototyping of avionic architecture specifications [19]. AADL is a SAE standard aimed at the high level design and evaluation of the architecture of embedded systems. Our aim is to interpret AADL specifications in the synchronous model of computation of Polychrony in order to provide a framework for the simulation, test and verification of integrated modular avionics. We use the existing techniques and library of the Polychrony environment, which consist of a model of the APEX-ARINC-653 real-time operating system sevices. A set of rules is defined for this transformation. We are currently working on the automatic translation using ATL as a model transformation language to describe the desired translation from AADL metamodel to Signal metamodel, and some test cases (such as avionics application examples) will be used for testing. Future work will focus on abstracting some general rules for the automatic code distribution from AADL architecture, and validate our model by the case study of an autonomous robot.

6.9. Modular compilation of imperative synchronous programs

Participants: Eric Vecchie, Jean-Pierre Talpin.

Although compilation of imperative synchronous languages like Esterel has been widely studied, the separate compilation of synchronous modules remains a challenge. We developed a compilation method inspired in traditional sequential code generation techniques applied to each source module. A salient difference is that we produce coroutines [32] arranged hierarchically to reflect the source control structure: our execution scheme is indeed inspired by the reactive kernel of Junior [34], a tool of the same Esterel family of language used to write reactive programs using the Java API. In Junior, each reaction step executes and reduces a tree representing the instantaneous state of the reactive program and reflecting its original structure. It suffices a minimalistic runtime system to execute our compiled module as "black boxes". As this compilation is structural and follows the State Behavioral Semantics of Esterel we should be able to prove that the generated code is correct with respect to this semantics. Last, but not least, our proposal is also applicable to add and compile higher-order synchronous modules in Esterel. This way, a module can be transmitted and instantiated through signals. To deal with constructivity issues, we defined and proved the correctness of a type system which reflects the potential emissions of a module. These works have been implemented as a lightweight compiler and will be available as a free software soon.

6.10. Virtual prototyping of imperative programs

Keywords: *SSA*, *SystemC*.

Participants: Loïc Besnard, Thierry Gautier, Jean-Pierre Talpin.

The Espresso project team has pursued his effort on importing C components into Signal [35], in the project FotoVP. The aim of the FotoVP is to develop virtual prototyping tools and techniques for the simulation and verification of system-level C/SystemC specifications in synchronous models of computation.

This work is based on the GCC compiler starting from its version 4. The compiler takes an arbitrary C source code, does the compilation, and outputs a tree called GNU TREE SSA (SSA - *Single Statement Assignation*) which represents the original C source code in a sort of high level assembler instructions. The importer takes the resulting tree and produces Signal code. This approach is convenient for reusing the many existing C components and performing static code verification by the means of the Signal compiler.

This year a (new) prototype has been written. It runs for a subset of C (C without pointers). It must be optimized for the loops: currently, when a loop is detected in the C code, it produces a Signal program with an oversampling. A possible optimization could be to transform patterns into spatial iterations. Such an optimization could be represented by a Signal graph.

7. Contracts and Grants with Industry

7.1. Network of excellence Artist2

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

The Espresso project-team participates to the Artist2 network of excellence. Detailed presentations on the aim and scope of the network can be found in the book [1] and the website http://www.artist-embedded.org/FP6 of the project. In particular, we have contributed to a survey of real-time programming languages edited by Alan Burns [17].

7.2. RNTL project OpenEmbeDD

Participants: Christian Brunette, Jean-Pierre Talpin.

The Espresso project-team coordinates (in collaboration with project Triskell) the RNTL project OpenEmbeDD. The goal of OpenEmbeDD is to develop an open-source toolset consisting of:

- model-driven design infrastructures
- asynchronous design and verification environments
- synchronous design and verification environments

for the design of embedded software. The project comprises many industrial partners to carry out case studies and validate the design environment. In the frame of OpenEmbeDD, we develop Eclipse plugins for Polychrony in collaboration with Airbus (Topcased technology) and project Atlas (ATL technology).

7.3. ANR project Topcased

Participants: Loïc Besnard, Paul Le Guernic, Julio Peralta, Yue Ma, Jean-Pierre Talpin.

The Espresso project-team participates to the Topcased initiative of Airbus. The aim of the Topcased initiative is to develop an open-source toolset for the design of avionic architectures. A summary of Topcased appears in [41]. The Topcased project is funded by the ANR and the Midi-Pyrénée region.

7.4. RNTL project Spacify

Participants: Loïc Besnard, Julien Ouy, Jean-Pierre Talpin.

The Espresso project-team participates to the RNTL project Spacify, leaded by CNES and ONERA. Spacify is a research project aiming at developing a design environment for spacecraft flight software.

More precisely, the project shall promote a top-down method based upon multi-clock synchronous modeling, formally-verified transformations, exhaustive verification through model-checking and a runtime framework featuring realtime-friendly distribution and dynamic-reconfiguration services. Furthermore, the various tools shall be released under FLOSS (free/libre/open-source software) licenses, favouring cost-sharing and sustainability.

The project is led by the French Agences CNES and ONERA. It gathers prime contractors Astrium Satellites and Thales Alenia Space, GeenSys (formerly TNI Software) and Anyware Technologies, and academic teams Cama from TELECOM Bretagne, Espresso from INRIA, MV from LaBRI and Acadie from IRIT. It is a 3-years project (starting in February 2006) partly funded by the French Research Agency (ref. ANR 06 TLOG 27).

7.5. ANR project FotoVP

Participants: Loïc Besnard, Thierry Gautier, Jean-Pierre Talpin, Eric Vecchie.

The Espresso project-team participates to the ANR project FotoVP, leaded by Verimag. The aim of the FotoVP is to develop virtual prototyping tools and techniques for the simulation and verification of system-level C/SystemC specifications in synchronous models of computation.

7.6. EADS Grant

Participants: Yann Glouche, Jean-Pierre Talpin.

The Espresso project-team received a grant from the EADS Foundation to fund a Doctorate on contract-based design in a polychronous model of computation. The aim of this program is to develop a model-driven engineering framework, based on the Eclipse plugins for Polychrony (developed in the frame of 7.2), allowing for the seamless integration of heterogeneous embedded system components within a contract-based design MDD environment.

8. Dissemination

8.1. Advisory

- Jean-Pierre Talpin is external advisory board member of the center of embedded systems at Virginia Tech, steering committee member of the ACM-IEEE conference on methods and models for codesign (MEMOCODE), steering committee member of the SLAP++ workshop series, organization committee member of the FMGALS workshop series, and editorial board member of the EURASIP Journal on Embedded Systems.
- Loïc Besnard served as President in a Jury for the competitive selection of engineers at CNRS.

8.2. Conferences

- Jean-Pierre Talpin served as technical program committee member for the conferences MEMOCODE'08, ACSD'08, EMSOFT'08 and SAC'08. He served as Guest Editor for two special sections of the IEEE Transactions on Computing and the IEEE Transactions in Industrial Informatics.

8.3. Thesis

- Jean-Pierre Talpin participated as referee to the Thesis defense of Gwenael Delaval at INRIA Grenoble on July 1st.

8.4. Teaching

- Thierry Gautier and Loïc Besnard taught on real-time programming at the DIIC 2 Graduate program of the University of Rennes I.
- Thierry Gautier taught on formal methods for component and system synthesis at the Master 2 Graduate program of the University of Rennes 1.
- Hugo Metivier taught programming scientific language at the Licence 1 PCGI Graduate program, taught web design at the Licence 3 MIAGE and taught oriented object language at the DIIC 1 (Licence 3 of Engineering school) Graduate program.
- Yann Glouche taught on functionnal programming at the STPI Graduate program of the INSA of Rennes.

8.5. Visits

- With the support of the University of Rennes I and of INRIA, Sandeep Shukla (Virginia Tech) is visiting team Espresso from July 2008 until April 2009.

[17], [18], [21]

9. Bibliography

Major publications by the team in recent years

[1] B. BOUYSSOUNOUSE, J. SIFAKIS (editors). *Embedded Systems Design. The ARTIST Roadmap for Research and Development*, Thierry Gautier, contributor, Springer, Lecture Notes in Computer Science, Vol. 3436, 2005.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] A. GAMATIÉ, T. GAUTIER. *Synchronous Modeling of Avionics Applications using the SIGNAL Language*, in "Proceedings of the 9th IEEE Real-time/Embedded technology and Applications symposium (RTAS'03), Washington D.C., USA", IEEE Press, May 2003.

[7] A. GAMATIÉ, T. GAUTIER, P. LE GUERNIC, J.-P. TALPIN. *Polychronous Design of Embedded Real-Time Applications*, in "ACM Transactions on Software Engineering and Methodology (TOSEM)", 2006.

[8] A. GAMATIÉ, T. GAUTIER, P. LE GUERNIC, J.-P. TALPIN. *Polychronous Design of Embedded Real-Time Applications*, in "ACM Transactions on Software Engineering and Methodology (TOSEM)", 2007.

[9] 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.

[10] 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.

[11] 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.

[12] 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^o 9, Septembre 1991, p. 1321–1336.

[13] 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.

[14] 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^o 4, October 2000, p. 347–368.

[15] J.-P. TALPIN, P. LE GUERNIC. *An algebraic theory for behavioral modeling and protocol synthesis in system design*, in "Formal Methods in System Design", 2006.

Year Publications

International Peer-Reviewed Conference/Proceedings

[16] A. GAMATIÉ, T. GAUTIER, L. BESNARD. *An Interval-Based Solution for Static Analysis in the SIGNAL Language*, in "Fifteenth IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS 2008), Proceedings, Belfast, Northern Ireland", IEEE Computer Society, 2008, p. 182-190.

[17] T. GAUTIER, P. LE GUERNIC, J.-P. TALPIN. *Polychronous Design of Real-Time Applications with SIGNAL*, in "ARTIST Survey of Programming Languages", A. BURNS (editor), http://www.artist-embedded.org/artist/ARTIST-Survey-of-Programming.html.

[18] B. JOSE, S. SHUKLA, H. PATEL, JEAN-PIERRE. TALPIN. *On the automatic inference of synchronization logic for multi-threaded software synthesis from polychronous specifications*, in "ACM-IEEE Conference on Methods and Models for Codesign", IEEE, 2008.

[19] Y. MA, JEAN-PIERRE. TALPIN, THIERRY. GAUTIER. *Virtual prototyping AADL architectures in a polychronous model of computation*, in "ACM-IEEE Conference on Methods and Models for Codesign", IEEE, 2008.

[20] H. METIVIER, J.-P. TALPIN, T. GAUTIER, P. LE GUERNIC. *Analysis of periodic clock relations in polychronous systems*, in "IFIP, Distributed Embedded Systems: Design, Middleware and Ressources (DIPES'08), Milano", vol. 271, september 2008.

Research Reports

[21] Y. GLOUCHE, P. LE GUERNIC, J.-P. TALPIN, T. GAUTIER. *A boolean algebra of contracts for logical assume-guarantee*, Technical report, n^{o} RR-6570, INRIA, 2008, http://hal.inria.fr/inria-00292870/fr/.

References in notes

[22] *OpenEmbeDD website*, http://openembedd.org.

[23] *Polychrony Update Site for Eclipse plug-ins*, http://www.irisa.fr/espresso/Polychrony/update/.

[24] *TopCased website*, http://www.topcased.org.

[25] AIRLINES ELECTRONIC ENGINEERING COMMITTEE. *ARINC Report 651-1: Design Guidance for Integrated Modular Avionics*, Technical report, Aeronautical radio, Inc., Annapolis, Maryland, 1997.

[26] AIRLINES ELECTRONIC ENGINEERING COMMITTEE. *ARINC Specification 653: Avionics Application Software Standard Interface*, Technical report, Aeronautical radio, Inc., Annapolis, Maryland, 1997.

[27] 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.

[28] B. BERTHOMIEU, P.-O. RIBET, F. VERNADAT. *The tool TINA - construction of abstract state spaces for Petri Nets and Time Petri Nets*, in "International Journal of Production Research", vol. 42, n^{o} 14, 2004.

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

[30] J. BUCK, S. HA, E. A. LEE, D. G. MESSERSCHMITT. *Ptolemy: A Framework for Simulating and Prototyping Heterogenous Systems*, in "Int. Journal in Computer Simulation", vol. 4, n^{o} 2, 1994, p. 155-182, http://citeseer.ist.psu.edu/buck92ptolemy.html.

[31] J.-L. COLACO, B. PAGANO, M. POUZET. *A conservative extension of synchronous data-flow with state machines*, in "In Embedded Software Conference.", ACM Press, 2005.

[32] M. CONWAY. *Design of a separable transition-diagram compiler*, in "Communications of the ACM", vol. 6, n^{o} 7, July 1963.

[33] H. GARAVEL, F. LANG, R. MATEESCU, W. SERWE. *CADP 2006: A Toolbox for the construction and Analisys of Distributed Processes*, in "Proceedings of the 19th International Conference on Computer Aided Verification", vol. 4590, Springer, 2007.

[34] L. HAZARD, J.-F. SUSINI, F. BOUSSINOT. *The Junior Reactive Kernel*, Research Report, n^{o} 3732, INRIA, July 1999.

[35] H. KALLA, J.-P. TALPIN, D. BERNER, L. BESNARD. *Automated translation of C/C++ programs into a synchronous formalism*, in "Engineering of Computer Based Systems, IEEE Press", March 2006.

[36] F. MARANINCHI, Y. RÉMOND. *Mode-automata: a new domain-specific construct for the development of safe critical systems*, in "Sci. Comput. Program.", vol. 46, n^{o} 3, 2003, p. 219–254.

[37] K. L. MCMILLAN. *Symbolic Model Checking: An approach to the state explosion problem*, Ph. D. Thesis, Carnegie Mellon University, May 1992.

[38] P. S. PACHECO. *A User's guide to MPI*, Technical report, Department of Mathematics, Univerity of San Francisco.

[39] E. RUTTEN, F. MARTINEZ. *Signal GTI: implementing task preemption and time intervals in the synchronous data flow language Signal*, in "Proceedings of the 7th Euromicro Workshop on Real-Time Systems, Odense, Denmark", IEEE Publ., june 1995.

[40] J.-P. TALPIN, C. BRUNETTE, T. GAUTIER, A. GAMATIÉ. *Polychronous mode automata*, in "Embedded Software Conference, ACM Press", September 2006.

[41] F. VERNADAT, C. PERCEBOIS, P. FARAIL, R. VINGERHOES, A. ROSSIGNOL, J.-P. TALPIN, D. CHEMOUIL. *The Topcased project - a toolkit in open-source for critical application and system development*, in "International Space System Engineering Conference, Eurospace", May 2006.