Component-based engineering of real-time JAVA : applications on a polychronous design platform

Jean-Pierre Talpin, Bruno Le Dez, Abdoulaye Gamatié, Paul Le Guernic, David Berner

INRIA
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Component-based engineering of real-time JAVA applications on a polychronous design platform

Jean-Pierre Talpin, Bruno Le Dez, Abdoulaye Gamatié, Paul Le Guernic, David Berner

N˚4744

February 2003

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4744--FR+ENG

RENNES

Component-based engineering of real-time JAVA* applications on a polychronous design platform†

Jean-Pierre Talpin, Bruno Le Dez, Abdoulaye Gamatié, Paul Le Guernic, David Berner

Thème 1 — Réseaux et systèmes
Projet ESPRESSO

Rapport de recherche n° 4744 — February 2003 — 21 pages

Abstract: Rising complexity and performances of embedded systems, shortening time-to-market demands for digital equipments, growing installed bases of intellectual properties, stress high-level design as a prominent research topic to compensate a widening productivity gap. In this aim, we put the principles of polychronous design (i.e. multi-clocked and synchronous) to work in the context of the real-time JAVA programming language by introducing a method for modeling, transforming, verifying and simulating multi-threaded real-time JAVA components in the multi-clocked synchronous data-flow design language SIGNAL. We put this technique to work for the component-based design and refinement of an even-parity checker with communication protocols.

Key-words: polychronous model of computation, component-based engineering, embedded system design, real-time JAVA

(Résumé : tsvp)

* JAVA is a registered trademark of SUN microsystems
† Work funded by the RNTL project EXPRESSO

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Component-based engineering of real-time JAVA apllications on a polychronous design platform

Résumé : L'ingénierie d'applications embarquées en avionique passe par plusieurs étapes de conception allant du prototypage sur station de travail, à la réalisation de simulateurs puis le déploiement de composants logiciels sur des architectures embarquées. La plate-forme POLYCHRONY permet d'aider à la conception d'application dans ce flot de conception en permettant la capture en amont de modélisation de haut niveau (par exemple en Java temps réel) et leur spécialisation, c'est-à-dire la production, correcte par construction, d'exécutifs temps réel, en utilisant les techniques formelles mises en œuvre dans la plate-forme POLYCHRONY.

Mots-clé :

1 Introduction

Rising complexity and performances of embedded systems, shortening time-to-market demands for digital equipments, growing installed bases of intellectual properties, stress high-level design as a prominent research topic and call for appropriate methodological solutions. In this aim, system design based on the so-called "synchronous hypothesis" has gained popularity. Synchronous programming consists of abstracting non-functional implementation details of a system away and of a focused design on the logics behind the instants at which the system functionalities should be secured. From this point of view, synchronous design models [11] and languages [6] provide intuitive models for embedded system design. In the relational model of the SIGNAL/POLYCHRONY design language/platform [11, 15] this affinity goes beyond the domain of purely synchronous systems to embrace the context of architectures consisting of synchronous circuits and desynchronized protocols: GALS architectures. The unique features of this model are to provide the notion of *polychrony*: the capability to describe multi-clocked (or partially clocked) circuits and systems and to support formal design *refinement* from the early stages of requirements specification to the later stages of synthesis and deployment and by using formal verification techniques. In the present article, we put the principles of polychronous design to work in the context of a functional subset of the real-time JAVA programming language [17] (compliant with avionics software certification requirements, e.g., no dynamic resources allocation) by introducing a method for translating and modeling multi-threaded real-time JAVA programs. We put this modeling technique to work by studying the refinement of an even-parity checker (EPC) toward its distributed implementation. Our goal is to derive automatically verifiable conditions on specifications under which refinement-based design principles work. In other words, we seek toward tools and methodologies to allow to take a high-level system component descriptions and to refine them in a semantic-preserving manner into GALS implementations.

2 An introduction to SIGNAL

In SIGNAL, a process P consists of simultaneous equations over signals. A signal $x \in \mathcal{X}$ describes a possibly infinite flow of discretely-timed values $v \in \mathcal{V}$. An equation $\boldsymbol{x} = f\boldsymbol{y}$ denotes a relation between a sequence of operands $\boldsymbol{y}$ and a sequence of results $\boldsymbol{x}$ by an operator f. Synchronous composition $P \parallel Q$ consists of the simultaneous solution of the equations P and Q in time. SIGNAL requires three primitive operators: pre references the previous value of a signal in time (the equation $x = \mathsf{pre}\, v\, y$ or $x = y\$1\,\mathsf{init}\, v$ initially defines x by v and then by the previous value of y in time), when samples a signal (the equation $x = y$ when z defines x by y when z is true) and default merges two signals (the equation $x = y$ default z defines x by y when y is present and by z otherwise).

$$P ::= \boldsymbol{x} = f\,\boldsymbol{y} \mid P \mid Q \mid P\,\mathsf{where}\,x \qquad f \in F \supseteq \{\,\mathsf{pre}\,v \mid v \in \mathcal{V}\} \cup \{\,\mathsf{when}\,,\,\mathsf{default}\,,\,\mathsf{not}\,,\,\mathsf{eq}\,,\,\ldots\}$$

A first example. We consider the definition of a counter: Count. It accepts an input event rst and delivers the integer output val. A local variable cnt, initialized to 0, stores the previous value of val (equation cnt := val$1 init 0). When the event rst occurs, val is reset to 0 (i.e. 0 when rst). Otherwise, cnt is incremented (i.e. (cnt + 1)). The activity of Count is governed by the clock of its output val which differs from that of its input rst.

```
process Count = (? event rst ! integer val)
  (| cnt := val$1 init 0
   | val := (0 when rst) default (cnt + 1)
   |) where integer cnt end;
```

time	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
rst		tt					tt				tt	tt
val	1	0	1	2	3	4	0	1	2	3	0	0
cnt	0	1	0	1	2	3	4	0	1	2	3	0

A model of polychronous signals. Starting from the models of computations of Lee et al. [10], we define the model of polychronous signals [11] for the formal study of protocol properties. We consider a set of boolean and integer *values* $v \in \mathcal{V}$ to represent the operands and results of a computation. A tag $t \in \mathbb{T}$ denotes an instant. The dense set $\mathbb{T}$ is equipped with a *partial order* relation $\leq$ to denote synchronization and causal relations. The subset $\mathcal{T} \subset \mathbb{T}$ of instants at which a given process is sampled is a countable semi-lattice $(\mathcal{T}, \leq, 0)$.

Definition 1 *An* event $e \in \mathcal{E} = \mathcal{T} \times \mathcal{V}$ *relates a tag and a value. A* signal $s \in \mathcal{S} = \mathcal{T} \rightharpoonup \mathcal{V}$ *is a partial function relating a* chain *of tags to a set of values. A* chain $C \in \mathcal{C}$ *is a totally ordered subset of* $\mathbb{T}$. *We write* $\text{tags}(s)$ *for the domain of a signal* s. *A* behavior $b \in \mathcal{B} = \mathcal{X} \rightharpoonup \mathcal{S}$ *is a partial function from signal names* $x \in \mathcal{X}$ *to signals* $s \in \mathcal{S}$. *We write* $\text{vars}(b)$ *for the domain of* b *and* $\text{tags}(b) = \cup_{x \in \text{vars}(b)} \text{tags}(b(x))$ *for its tags. Hence, the informal sentence "*x *is present at* t *in* b*" is formally defined by* $t \in \text{tags}(b(x))$. *A* process $p \in \mathcal{P}$ *is a set of behaviors that have the same domain* X *(written* $\text{vars}(p)$*).*

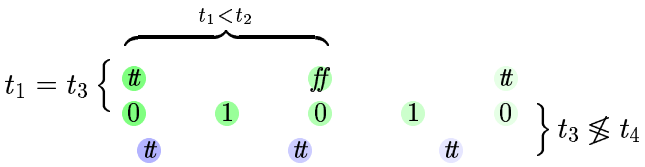


Figure 1: A behavior b: from signal names to partially ordered tags and values

Synchronous composition. $p \| q$ is defined by the set of behaviors that extend a behavior $b \in p$ by the restriction $c_{/\text{vars}(p)}$ of a behavior $c \in q$ if the projections of b and c on $\text{vars}(p) \cap \text{vars}(q)$ are equal. We note $I = \text{vars}(p) \cap \text{vars}(q)$, write $b|_X$ for the projection of a behavior b on $X \subset \mathcal{X}$ (i.e. $\text{vars}(b|_X) = X$ and $\forall x \in X, b|_X(x) = b(x)$) and define $b_{/X}$ as $b|_{\text{vars}(b) \setminus X}$.

$$p \| q = \{ b \cup c \,|\, (b, c) \in p \times q, b|_I = c|_I \}$$

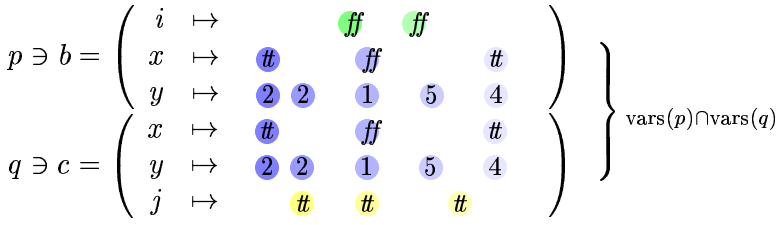


Figure 2: Synchronous composition $p \| q$: matching behaviors along common signals

Scalability. This is a key concept for engineering systems and reusing components in a smooth design process. In our model, support for scalability is provided by the *stretch-closure property*. The intuition behind this relation is to consider a signal as an elastic with ordered marks on it (tags). If it is stretched, marks remain in the same (relative and 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 partial order between marks is unchanged: timing (synchronization and causal) relations are preserved. Formally,

Definition 2 *A behavior c is a* stretching *of b, written $b \leq c$, iff* $\text{vars}(b) = \text{vars}(c)$ *and there exists a bijection f on $\mathcal{T}$ that is strictly increasing, monotonic along all chains and satisfies* $\text{tags}(c(x)) = f(\text{tags}(b(x)))$ *for all $x \in \text{vars}(b)$ and $b(x)(t) = c(x)(f(t))$ for all $x \in \text{vars}(b)$ and all $t \in \text{tags}(b(x))$. The behaviors b and c are* stretch-equivalent*, written $b \lesseqgtr c$, iff there exists a behavior d s.t. $d \leq b$ and $d \leq c$. A process p is* stretch-closed *iff for all $b \in p$, $c \lesseqgtr b \Rightarrow c \in p$. A non-empty, stretch-closed process p admits a set of strict behaviors, written $(p)_{\lesseqgtr}$, s.t. $(p)_{\lesseqgtr} \subset p$ (for all $b \in p$, there is a unique $c \in (p)_{\lesseqgtr}$ s.t. $c \lesseqgtr b$).*

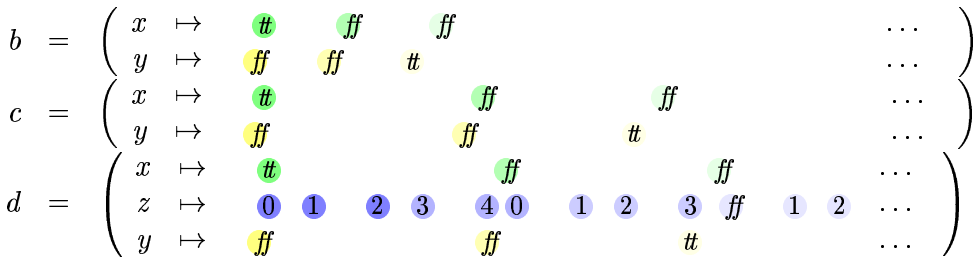


Figure 3: Stretching b allows for a scalable c and refinement d

Asynchrony. We consider a weaker relation (which discards synchronization relations) for comparing behaviors w.r.t. the sequences of values signals hold. The *relaxation* relation allows to individually stretch the signals of a behavior. A behavior c is a *relaxation* of b, written $b \sqsubseteq c$, iff $\text{vars}(b) = \text{vars}(c)$ and for all $x \in \text{vars}(b)$, $b_{|x} \leq c_{|x}$. Relaxation is a partial-order relation that defines the flow-equivalence relation. Two behaviors are flow-equivalent iff their signals hold the same values in the same order. The behaviors b and c are *flow-equivalent*, written $b \approx c$, iff there exists a behavior d s.t. $b \sqsubseteq d \sqsupseteq c$. Flow-equivalence defines is a semi-lattice: a behavior b admits a strict behavior, written $(b)_{\approx}$. We use relaxation to define asynchronous composition (we note $I = \text{vars}(p) \cap \text{vars}(q)$):

$$p \parallel q = \{d \,|\, \exists (b, c) \in p \times q, b|_I \sqsubseteq d|_I \sqsupseteq c|_I\}$$

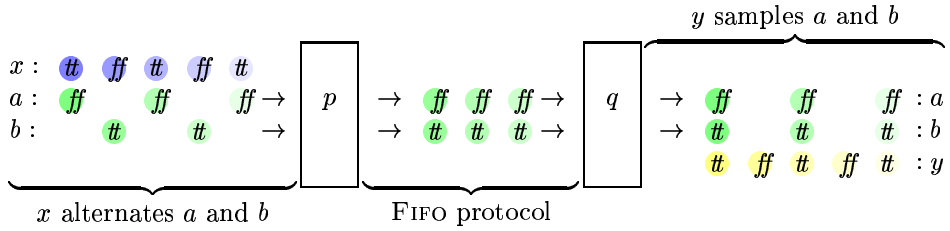


Figure 4: Asynchrony as the relaxation of synchronization relations

Formal design methodology. The model of polychrony provides a purely relational denotational semantics of SIGNAL (defined by the function $[\![P]\!]$ below) that defines the set of possible behaviors of a SIGNAL process p. The denotation defines the relation between events along the tag chains of signals involved in an equation. $[\![x = \text{pre}\, v\, y]\!]$ defines the signals x and y along a chain of tags $C \in \mathcal{C}$: the value of x at t is that of y for the immediate predecessor of t in C (notice that x and y are synchronous). $[\![x = y \,\text{when}\, z]\!]$ defines x by y when z is true i.e. if $t \in \text{tags}(y) \cap \text{tags}(z)$ and

$z(t) = tt$ then $t \in \text{tags}(x)$ and $x(t) = y(t)$, otherwise $t \notin \text{tags}(x)$. $[\![x = y \text{ default } z]\!]$ defines x by y when y is present and by z otherwise.

$$
\begin{array}{rl}
[\![x := \text{pre}\, v\, y]\!] = & \left\{ b \in \mathcal{B}|_{x,y} \;\middle|\; \begin{array}{l} \text{tags}(b(x)) = \text{tags}(b(y)) = C \in \mathcal{C} \setminus \emptyset,\ b(x)(\min(C)) = v \\ \forall t \in C \setminus \min(C),\ b(x)(t) = b(y)(\text{pred}_C(t)) \end{array} \right\} \cup \{0|_{x,y}\} \\
[\![x := y \text{ when } z]\!] = & \left\{ b \in \mathcal{B}|_{x,y,z} \;\middle|\; \begin{array}{l} \text{tags}(b(x)) = \{t \in \text{tags}(b(y)) \cap \text{tags}(b(z)) \mid b(z)(t) = tt\} \\ \forall t \in \text{tags}(b(x)),\ b(x)(t) = b(y)(t) \end{array} \right\} \\
[\![x := y \text{ default } z]\!] = & \left\{ \{b \in \mathcal{B}|_{x,y,z} \;\middle|\; \begin{array}{l} \text{tags}(b(y)) \cup \text{tags}(b(y)) = \text{tags}(b(x)) = C \in \mathcal{C} \\ \forall t \in C,\ b(x)(t) = \text{if } t \in \text{tags}(b(y)) \text{ then } b(y)(t) \text{ else } b(z)(t) \end{array} \right\}
\end{array}
$$

The semantics of SIGNAL is closed in the structure of polychronous signals: whenever a process P (resp. network Q) has a behavior b, written $b \in [\![P]\!]$, then it admits any stretching $c \geq b$ (resp. relaxation $c \sqsupseteq b$) of b, i.e. $c \in [\![P]\!]$.

$$
\begin{array}{rcl}
[\![x = \text{pre}\, tt\, y]\!] & \ni & \begin{pmatrix} y & \mapsto & ff & tt & ff & tt & \dots & \\ x & \mapsto & tt & ff & tt & ff & tt & \dots \end{pmatrix} \\
[\![x = y \text{ default } z]\!] & \ni & \begin{pmatrix} y & \mapsto & ff & & ff & tt & & \dots \\ z & \mapsto & tt & ff & & ff & tt & \dots \\ x & \mapsto & ff & ff & ff & tt & tt & \dots \end{pmatrix} \\
[\![x = y \text{ when } z]\!] & \ni & \begin{pmatrix} y & \mapsto & tt & tt & & & tt & \dots \\ z & \mapsto & ff & & ff & tt & tt & \dots \\ x & \mapsto & & & & & tt & \dots \end{pmatrix}
\end{array}
$$

Figure 5: Delay, sampling and merge

Formal design properties. The model of polychronous signals allows to define formal properties that are essential for the component-based design of GALS architectures [11]. *Endochrony* is a key design property. A process is endochronous iff, given an external (asynchronous) stimulation of its inputs I, it reconstructs a unique synchronous behavior (up to stretch-equivalence). Endochrony denotes the class of processes that are insensitive to (internal and) external propagation delays.

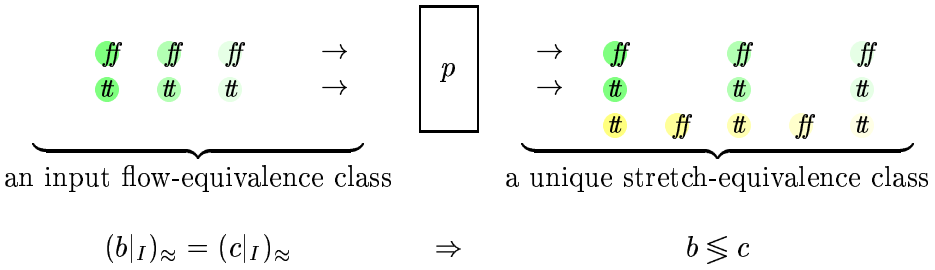


Figure 6: An endochronous design

Flow-equivalence offers the right criterion for checking the refinement of a high-level system specification with distributed communication protocols correct. For instance, it is considered in [5] for the refinement-based design of the LTTA protocol in SIGNAL. *Flow-invariance* is the property that ensures that the refinement of a functional specification $p \,|\, q$ by an asynchronous implementation $p \parallel q$ preserves flow-equivalence. Formally,

Definition 3 *A process p is endochronous on its input signals I iff $\forall b, c \in p$, $(b|_I)_\approx = (c|_I)_\approx$ implies $b \leqslant c$ (clock equivalence). The designs p and q are flow-invariant iff, for all $b \in p \| q$, for all $c \in p \parallel q$, $(b|_I)_\approx = (c|_I)_\approx$ implies $b \approx c$ (flow equivalence) for I the inputs of $p \| q$.*

Clocks and causality relations. Distributed protocol synthesis and sequential code generation (to, e.g., ANSI C, JAVA, VHDL) in SIGNAL are ultimate design stages performed, from a given system model (e.g. a modulo 3 counter), after an analysis stage: the inference of synchronization (e.g. signals i, s and o are synchronous) and scheduling relations (e.g. c and o cannot happen before s when i is present); and a verification and optimization stage: the construction of a canonical control flow graph starting from these data [3]. The generated code that corresponds to this canonical control-flow graph has minimal branching. Automated distribution using protocol synthesis techniques are implemented in POLYCHRONY [4].

model	clocks	scheduling	code
s:=o\$1 init 2	$s \hat{=} o$	$\emptyset$	if i then { c = (s == 2);
| c:=when ($s = 2$)	$c \hat{<} s$	$s \rightarrow^i c$	if c then o = 0 else o = s + 1;
| o:=(0 when c) default $s + (1$ when $i)$	$o \hat{=} s \hat{=} i$	$s \rightarrow^i o$	s = o; }

3 A real-time JAVA plugin for POLYCHRONY

We put our modeling tool to work by studying the refinement of an even-parity checker (EPC) toward its distributed implementation. Our goal is to derive automatically verifiable conditions on specifications under which refinement-based design principles work. In other words, we seek toward tools and methodologies to allow to take a high-level system component descriptions and to refine them in a semantic-preserving manner into GALS implementations.

The component-based engineering of real-time JAVA classes using the SIGNAL platform POLYCHRONY (figure 7) consists of modeling the behavior of the underlying runtime system (the real-time JAVA virtual machine) and of translating the classes it is composed of, by instantiating the runtime model to the way it is used in the application, and by translating the real-time JAVA threads and event handler into the SIGNAL design language.

The benefits of using POLYCHRONY for engineering real-time JAVA classes lies in the formal methods provided by this backbone platform, using which non-trivial architecture refinements, protocol synthesis and correct-by-construction optimization techniques can automatically be applied.

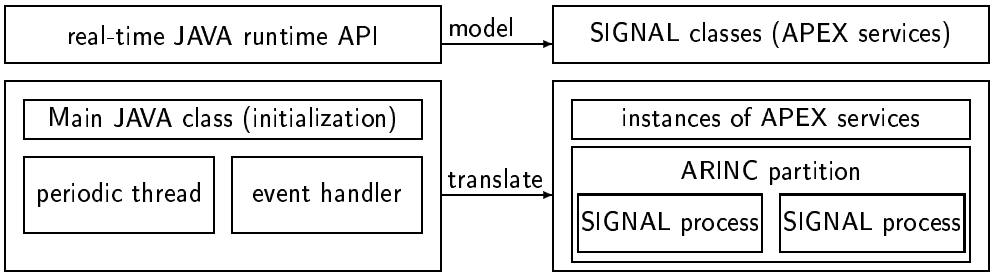


Figure 7: Architecture of the real-time JAVA plugin for POLYCHRONY

A case study

To illustrate this design refinement process, we focus on the study of an even-parity checker protocol (EPC, figrue 8), showing how the initial JAVA specification can be refined toward a GALS implementation with the help of POLYCHRONY, showing in what respects and at which critical design stages formal methods matter for engineering such architectures.

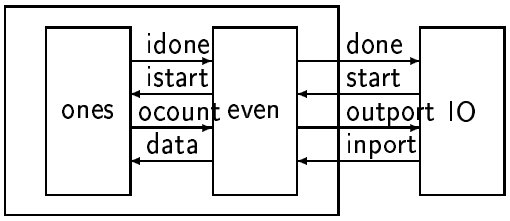


Figure 8: Architecture of an even-parity checker protocol

The EPC consists of three functional units: an IO interface process, an even test process and a main ones counting process.

```
import javax.realtime.*;
class parity {
    public static int inport, outport, data, ocount;
    public static boolean start, done, istart, idone;
    public static void main (String argv[]) {
        IO Io = new IO ();
        even Even = new even ();
        ones Ones = new ones ();
        Io.start();
        Even.start();
        Ones.start(); }}
```

A translation method

A real-time JAVA program can be translated into SIGNAL starting from either its source JAVA code or its byte-code. To this aim, we use the JAVA compilation tool SOOT [16] to pre-process classes and obtain an optimized control-flow graph and associated the intermediate representation: JIMPLE. JIMPLE is a very handy format to process JAVA classes. It consists of explicitly typed, stack-less, 3-address code statements (grammars *stm* and *rtn*) that manipulates either immediates (grammar i) or references (grammar v).

We write l for a local variable (assigned once in JIMPLE), c for a constant, t for a type name, L for a program label and x for a class field name. Declarations (grammar *dec*) may not occur in the run method of threads and event handler as they might dynamically allocate memory: they are assumed to be present in the main initialization class of the program, or the init method of thread classes (executed once at system start). In JIMPLE, declarations are explicitly typed and and initialized. Furthermore, monitor (un)locking is explicit (even in the presence of exceptions: monitors are released before exceptions are raised).

The translation of JIMPLE code distinguishes between the main (initialization) class and the init methods of other classes, which are scanned in order to determine the layout and architecture of the application (its shared data-structures and its thread parameters) and the run method of classes

that implement the periodic threads and event handlers of the system. The *run* sequence of a thread consists of a sequence of blocks *blk* that consist of a label L, a sequence of operations *stm* and a return statement *rtn*.

i	::=	$l \mid c$	(immediate)	*stm*	::=	$v = i \star i$	(statement)
$\star$	::=	$+ \mid - \mid \ldots$	(operator)		$\mid$	$v =$ invoke i (i^*)	(invocation)
v	::=	$i[i] \mid i.[x] \mid x \mid l$	(variable)		$\mid$	$l := [v \mid @r]$	(local)
r	::=	caughtexception	(reference)	*rtn*	::=	entermonitor i	(lock)
	$\mid$	parameter c	(parameter)		$\mid$	exitmonitor i	(unlock)
	$\mid$	this	(self)		$\mid$	goto L	(goto)
					$\mid$	if i then L	(test)
dec	::=	$v = i$ instanceof t	(instantiation)		$\mid$	return	(return)
	$\mid$	$v =$ new $t[i]$	(initialization)		$\mid$	throw i	(throw)
	$\mid$	$t\ x$	(declaration)		$\mid$	catch t from L	(catch)
run	::=	$blk \mid run; run$	(program)			to L	
blk	::=	$L : stm^*; rtn \mid blk; blk$	(block)			using L	

Modeling the real-time runtime system using APEX services.

The APEX interface, defined in the ARINC standard [2], 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, [1]). 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* (figure 9).

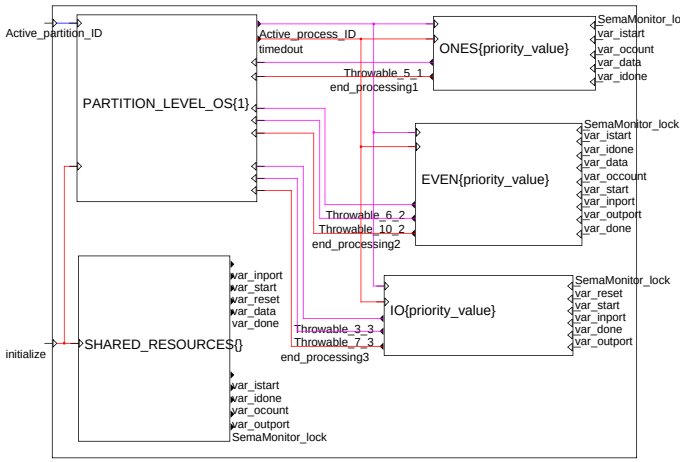


Figure 9: Architecture of the ARINC partition for the EPC model in SIGNAL (partition).

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 APEX interface includes both services to achieve communications and synchronizations, and services for the management of processes and partitions. A model of the APEX services for process partitioning and management, for communication and synchronization, is implemented in SIGNAL (see [8]). It consists of generic, parameterizable SIGNAL modules which implement the APEX API. We use it to model (and implement) the real-time JAVA virtual machine. Figure 9 defines the instance of this model for the EPC case study.

Example: lock monitoring.

The model of a simplified lock monitoring protocol in SIGNAL consists of setting a lock to the status free upon request of the notifier. A waiting process checks whether the lock (initialized to 0 to mean free) has been notified at the previous instant (it is free) and is available at its own request clock. If so, the event granted is present and the lock is set to the owner block identifier. The actual semaphore mechanism of the APEX service library suspends processes waiting on a locked semaphore and resumes a waiting process upon a notification according to specified real-time scheduling policies.

In the example, we note $x ::= f(\boldsymbol{y})$ when c the partial definition of x by $f(\boldsymbol{y})$ at the clock c. Composed to $x ::= f(\boldsymbol{z})$ when d, it is equivalent to the equation $x := f(\boldsymbol{y})$ when c default $f(\boldsymbol{z})$ when d iff the clocks c and d are exclusive (this constraint can be made explicit is SIGNAL as $c\hat{}\#\, d$). Satisfaction of clock exclusion constraints is checked by the clock resolution engine of the SIGNAL compiler (meaning that the assignment to x is deterministic). We note $\boldsymbol{x} := f\{\boldsymbol{c}\}(\boldsymbol{y})$ for a call to a SIGNAL process of module f that takes the static parameters $\boldsymbol{c}$.

```
process entermonitor={integer block, integer lock}( ? event request ! event active)
        (| active := when request when ((owner = 0) default (owner = block))
         | lock ::= block when active | owner := lock$1 when request
         |) where integer owner;
process exitmonitor={integer block, integer lock}( ? event request ! )
        (| lock ::= 0 when (lock$1 = block) when request
         |);
```

Example: architecture of the EPC main.

To illustrate how a real-time JAVA architecture description (as specified in the main method of its top -level class) is analyzed and used to generate a SIGNAL instance of APEX services that models it, we consider the case of the EPC class parity.

The processing of the main method of the parity class starts with a linear analysis of its declarations and *dec* statements which produces a tree structure where each node consists of a SIGNAL data structure that renders the category of each of the items initialized in this method (shared data-structure, periodic or sporadic thread, event handler) together with its initialization parameters (size, real-time parameters, trigger and handler).

Once the main method is scanned, the translation of real-time threads and event handler starts, in order to determine the remaining architecture parameters from the init method of each class and the number of critical sections from the run method of each class.

Given this data, the architecture of the system is finalized and the APEX services instantiated. This yields the structure depicted in figure 10 for thread ones: a control process is connected to the partition-level scheduler and a computation process (figure 11).

The APEX library provides the data-structures that allow to manipulate and simulate the real-time parameters extracted from the original JAVA specification (appendix B).

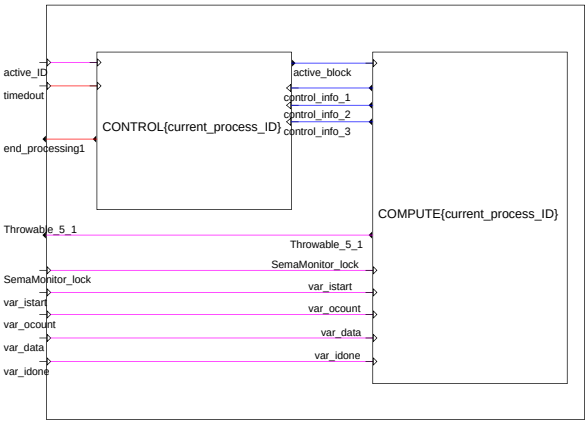


Figure 10: Architecture of the ARINC model of thread ones.

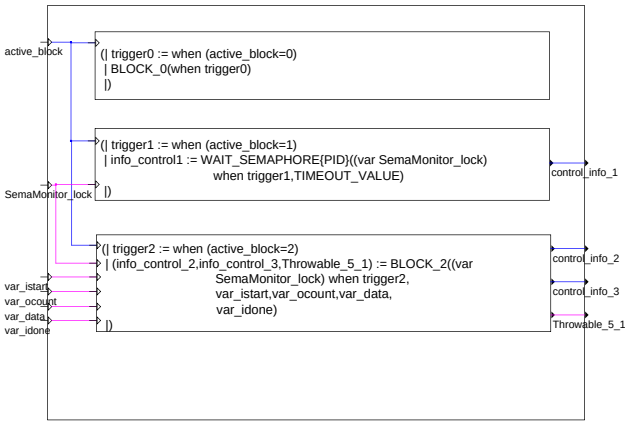


Figure 11: Computation block of the ARINC model of thread ones.

Translating periodic threads and event handlers in SIGNAL.

The translation of JIMPLE to SIGNAL is defined by the translation function $[\![run]\!]_E = \langle p \rangle$ and $[\![blk]\!]_E^C = \langle p \rangle$ which takes an input *run* method code as input, together with the activation clock of the target SIGNAL process, denoted by C. The *run* method is given an environment E which associates: method references r to local variables l (i.e. $\mathcal{R}_E(l) = r$); block and section labels L to activation clocks C (i.e. $\mathcal{C}_E(L) = C$); exceptions t raise at a label L to the corresponding exception flow (i.e. $\mathcal{X}_E(t, L) = (L_1, L_2)$, of target L_1 and handler L_2); a control-flow graph $\mathcal{G}_E(L)$ (i.e. $\text{main}(\mathcal{G}_E)$ gives the main entry label of *run* and $\text{pred}^*(\mathcal{G}_E)(L)$ all predecessors of label L in the graph).

$$[\![blk; run]\!]_E = \langle p \,|\, q \rangle \text{ where } [\![blk]\!]_E = \langle p \rangle \text{ and } [\![run]\!]_E = \langle q \rangle$$

A real-time periodic thread or an event handler is viewed as a sequence of critical sections that receives control from the partition-level scheduler via the tick clock and the next_block state variable (which ranges on the subset of labels, symbolically denoted by $\#L$, of interruptible blocks).

Sections A section consists of a sequence of elementary statements *stm* delimited by a label L and a return statement *rtn*. A section label defines the entry point for a given transition. Hence, it is the symbolic value of the global state variable next_block of use in the current APEX partition. A block label is denoted by an event: it is present iff the corresponding block is active during the current transition. Every statement of the block (computation *stm* or control *rtn*) is conditioned by that clock.

$$\begin{aligned}[\![L : stm_{1\ldots n}; rtn]\!]^C_E = \langle \mathcal{C}_E(L) ::= \mathsf{when}\, \mathsf{next\_block\$1} = \#L\, \mathsf{when}\, \mathsf{tick} \,|\, p_1 \,\|\, \ldots \,|\, p_n \,|\, p \rangle \\ \text{where for all } i = 1, \ldots n, [\![stm_i]\!]^C_E = \langle p_i \rangle \text{ and } [\![rtn]\!]^C_E = \langle p \rangle\end{aligned}$$

Statements A computation statement *stm* takes three forms. *1. The definition $l := r$ of a local variable l.* The use of local variables in JIMPLE code facilitate data-flow analysis. In the case of a location, it guarantees that the reference r is read and written once within a given block or section. The reference is translated to the previous value of the corresponding signal and the local variable is translated by a local (volatile) signal. In the case of a method, the reference is associated to the local l in the environment E of the translator.

$$\begin{aligned}[\![l := v]\!]^C_E &= \langle l ::= v\$1\, \mathsf{when}\, C \rangle \\ [\![l := @r]\!]^C_E &= \langle l ::= r\, \mathsf{when}\, C \rangle\end{aligned}$$

2. The call to an external method $v = \mathsf{invoke}\, i\, (i^*)$ (i.e. a method whose byte-code is not available to SOOT). All methods from available classes are inlined in the JIMPLE code of the thread/handler, in order to globally optimize its flow of control.

$$[\![v = \mathsf{invoke}\, i\, (i^*)]\!]^C_E = \langle v ::= \mathcal{R}_E(i)((i^*)\, \mathsf{when}\, C) \rangle$$

3. A pervasive operation $v = i \star i$ on immediate values i and j is directly translatable by the corresponding equation scheduled at the context clock C.

$$[\![v = i \star j]\!]^C_E = \langle v ::= (i \star j)\, \mathsf{when}\, C \rangle$$

Monitors Locks monitoring is modeled by the SIGNAL processes (entermonitor and exitmonitor).

$$\begin{aligned}[\![\mathsf{entermonitor}\, i]\!]^C_E &= \langle \mathsf{entermonitor}\{S, \mathcal{R}_E(i)\}(\mathsf{when}\, C) \rangle \\ [\![\mathsf{exitmonitor}\, i]\!]^C_E &= \langle \mathsf{exitmonitor}\{S, \mathcal{R}_E(i)\}(\mathsf{when}\, C) \rangle\end{aligned}$$

Control structures Control statements, such as gotos, ifs and throws consist of activating the clock that corresponds to the target block label.

$$\begin{aligned}[\![\mathsf{goto}\, L]\!]^C_E &= \mathrm{if}\, (L \notin \mathrm{pred}^*(\mathcal{G}_E)(L))\, \mathrm{then}\, \langle \mathcal{C}_E(L) ::= \mathsf{when}\, C \rangle \\ &\qquad \mathrm{else}\, \langle \mathsf{next\_block} ::= \#L\, \mathsf{when}\, C \rangle \\ [\![\mathsf{if}\, i\, \mathsf{then}\, L]\!]^C_E &= \mathrm{if}\, (L \notin \mathrm{pred}^*(\mathcal{G}_E)(L))\, \mathrm{then}\, \langle \mathcal{C}_E(L) ::= \mathsf{when}\, i\, \mathsf{when}\, C \rangle \\ &\qquad \mathrm{else}\, \langle \mathsf{next\_block} ::= \#L\, \mathsf{when}\, i\, \mathsf{when}\, C \rangle \\ [\![\mathsf{throw}\, i]\!]^C_E &= \langle \mathcal{C}_E(L_2) := \mathsf{when}\, C \,|\, \mathcal{C}_E(L_1) ::= \mathcal{C}_E(L_2)\, \mathsf{when}\, C \rangle \\ &\qquad \text{where } C = \mathcal{C}_E(L) \text{ and } \mathcal{X}_E(\mathcal{R}_E(i), L) = (L_1, L_2)\end{aligned}$$

In the particular case of the return statement, translation is performed by installing the corresponding pattern of APEX protocol at partition level (see, e.g. figure 9). We assume that variables v and references r are merely translated as is ($i.[x]$ as $i.x$ (a datum) or $i.x$ (a method), parameter c as parameter$c$, this by nothing, etc).

$$[\![\text{return}]\!]_E^C = \langle \text{next\_block} ::= \text{main}(\mathcal{G}_E) \text{ when } C \rangle$$

Example of the ones thread.

The concurrency model of the real-time JAVA starts at a design level where implicit architecture choices are already made: the system consists of a set of threads that interact via shared variables and locks). The thread ones determines the parity of an input data. Upon receipt of the start notification, ones shifts data until it 0 and the internal count is assigned to the output ocount and done notified. The thread even notifies ones to start processing data and waits until done is notified to read the final ocount and checks whether it is an even number.

The SIGNAL model of thread ones consists of one critical section, delimited by a pair of monitor statements. The process is activated when it obtains the lock on istart. Then, at its own rate (now conditioned by the clock c1), it determines the count. When it is finished, it sends the notification.

```
class ones extends PeriodicThread {
  ⋮
  public void run () {
    int data = 0, ocount = 0;
      synchronized (parity.lock) {
        if (parity.istart) {
          data = parity.data;
          ocount = 0;
          while (data != 0) {
            ocount = ocount + (data & 1);
            data = data >> 1; }
          parity.ocount = ocount;
          parity.idone = true;
          parity.istart = false; }}}}
```

```
process ones      = (? event tick ! )
(| c1             :=enter_monitor{1,parity_lock}(tick)
 | data           ::=parity_data when c1
 | ocount         ::=0 when c1
 | c2             :=when (data$1 <> 0) when tick
 | ocount         ::=(ocount$1 + ext2(data$1,1)) when c2
 | data           ::=ext1(data$1) when c2
 | c3             :=when (data$1 = 0) when tick
 | parity_ocount::=ocount when c3
 | parity_idone  ::=true when c3
 | parity_istart ::=false when c3
 |                   exit_monitor{1,parity_lock}(c3)
 |) where          integer data init 0, ocount init 0;
                   event c1, c2, c3 ;
```

Pervasive operators, e.g. the unsigned operators >> and &, are referenced as external functions.

```
function ext1 = ( ? i1 ! i2 )      spec (| i1 ^= i2 | i1→i2 |)
                                   pragmas JAVA_CODE "&i2 = &i1 >> 1" end pragmas;
function ext2 = ( ? i1, i2 ! i3 )  spec (| i1 ^= i2 ^= i3 | i1→i3 | i2→i3 |)
                                   pragmas J_CODE "&i3=&i1 & &i2" ... ;
```

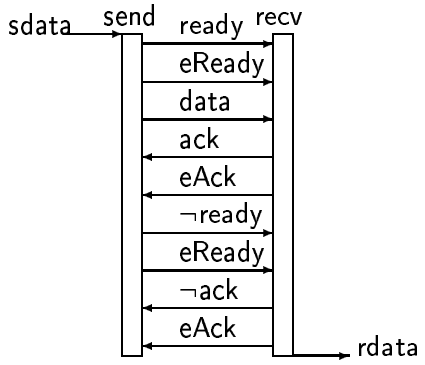
4 Checking system design refinements correct

With a real-time JAVA plugin at hand, we put our polychronous design methodology by experimenting with the refinement of the even-parity checker (EPC) from its initial specification toward its distributed implementation by the insertion of communicating protocols and by the merge of secondary threads on a given architecture. Our goal is to demonstrate how our polychronous design

tools and methodologies allow to take a high-level system component descriptions and to refine them in a semantic-preserving manner into GALS implementations.

Architecture design refinement: a double-handshake protocol.

We wish to model the physical distribution of the threads ones and even and allow them to communicate asynchronously via a channel structure that implements a double handshake protocol.



The model of send and recv methods in SIGNAL is obtained from its message sequence specification in a very same way as for the threads even and ones of the JAVA design, except that the ready and ack flags correspond to state variables (declared at the same lexical level as send in the channel module) and that eReady and eAck stand for events. By installing the channel process between producer and consumer, we obtain a desynchronization of the transmission between the ones and even processes.

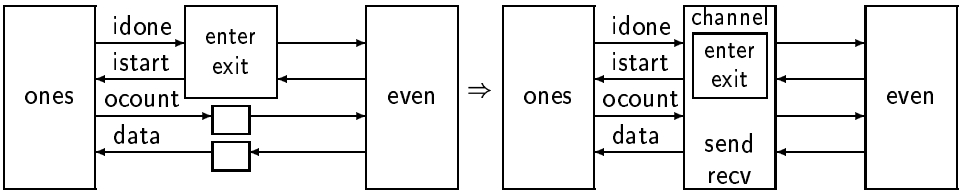
```
process send = (? integer sdata; event tick ! )
(| c1          :=when (event sdata) when tick
 | ready       ::=true when c1 | notify{eReady}(c1)
 | data        ::=sdata when c1
 | c2          :=when ack$1 when tick
 | c3          :=wait{eAck}(c2)
 | ready       ::=false when c3 | notify{eReady}(c3)
 | c4          :=when not(ack$1) when tick
 | c5          :=wait{eAck}(c4)
 |) where        event c1, c2, c3, c4, c5 ;
```

Sender and receiver use a simplified wait/notify mechanism similar to that implemented by the asynchronous event handlers in JAVA.

```
process notify ={boolean lock}( ? event tick ! ) (| lock ::= true when tick |);
process wait =  {boolean lock}( ? event tick ! event free)
                (| free := when lock$1 when tick | lock ::= false when tick |);
```

Validation of the protocol insertion.

Checking that the EPC model upgraded by a double-handshake protocol is a correct refinement of the initial model amounts to checking that the initial and upgraded designs are flow-equivalent ($\boxed{\ }$ stands for a register).



This amounts to proving that, for all behaviors b and c of the specification S_{ones} and of the architecture A_{ones} of the EPC, flow equivalence of the input signal inport, i.e. $b|_{\mathsf{inport}} \approx c|_{\mathsf{inport}}$ implies flow equivalence of the signal outport, i.e. $b|_{\mathsf{outport}} \approx c|_{\mathsf{outport}}$.

$$(1) : \forall b \in [\![S_{\mathsf{ones}}]\!],\ \forall c \in [\![A_{\mathsf{ones}}]\!],\ b|_{\mathsf{inport}} \approx c|_{\mathsf{inport}} \Rightarrow b|_{\mathsf{outport}} \approx c|_{\mathsf{outport}}$$

However, the refined model of the EPC only differs from the initial one by the introduction of a double handshake protocol, which implements the generic synchronization scheme S : data := (inport when c default data\$1) when clock of the JAVA specification. The matching pattern A : data := recv(clock) ‖ send(inport, clock) of the protocol in the architecture layer consists of the insertion of send and receive in place of this basic synchronization mechanism. Hence, proving equation (1) reduces to showing that the desynchronization protocol introduced by the channel process preserves flow equivalence between the original specification and the final architecture. This amounts to showing that the specification model S is flow-equivalent to the process A in the architecture model.

$$(2) : \forall b \in [\![S]\!],\ \forall c \in [\![A]\!],\ b|_{\mathsf{inport}} \approx c|_{\mathsf{inport}} \Rightarrow b|_{\mathsf{data}} \approx c|_{\mathsf{data}}$$

Notice that equation (2) structurally implies (1). Showing that A is flow-equivalent to S is amenable to symbolic model checking using the companion model-checking tool SIGALI [13] of SIGNAL. A general method for this type of proof consists of modeling an observer of property (2) in SIGNAL and then of using SIGALI to prove that its output signal never becomes false. The observer simulates the input inport using (arrays of) booleans (provided corresponding implementations of the external functions ext1 and ext2). A buffer is used to avoid altering synchronizing signals between the models S and A.

```
process observer = (? boolean i ! boolean invariant)
  (| invariant := buffer(S(buffer(i))) = buffer(A(buffer(i))) |)
where process buffer = ( ? boolean i ! boolean o)
          (| o := Current (i) | Alternate (i, o) |)
      process Current = (? boolean i ! boolean o)
        (| o := (i cell ^o init false) when ^o |)
      process Alternate = (? boolean i, o !)
        (| i ^= when flop | o ^= when not flop
         | flop := not (flop$1 init true)
         |) where boolean flop;
end;
```

Architecture design refinement: recombination of threads.

The encoding of the even-parity checker demonstrates the capability of SIGNAL to provide multi-clocked models of JAVA components for verification and optimization purposes. Polychrony allows for a better decoupling of the specification of the system under design from early architecture mapping choices. For instance, it allows for an optimized recombination of behaviors: the SIGNAL

compiler can merge the behaviors IO and even using its clock resolution engine. Checking the merge of the threads IO and even correct wrt. the initial specification amounts, first, to checking that it is endochronous ie. that it is able to reconstruct a unique flow of control given an external input. As shown in [11], this amounts to checking that the clock of its output data can be determined from that of its input data and from the master simulation tick (this shows that it is deterministic) and that the frequency of the output is lower than that of both inputs data and tick (this shows that control is hierarchical). Second, checking the merge of the threads IO and even correct wrt. the initial specification amounts to showing that the initial SIGNAL model of IO and even is flow equivalent to that of the merged threads evenIO in the same vein than for the specification refinement studied in the previous section.

Proving global design invariant.

In addition to the above methodology-specific verification issues, the companion model checker of SIGNAL allows to prove more general properties of specification requirements: reachability, attractivity and invariance (see [12] for details). Examples applications are, for instance, to check that the refinement of a model with a finite FIFO buffer satisfies requirements such as: "*one never reads an empty* FIFO *queue*" or "*one never writes to a full* FIFO *queue*". Such requirements have previously been studied in [9], in the context of the modeling of APEX avionics application in SIGNAL and the companion design methodologies. Another common requirements is non-interference: eg. a lock is never requested from two concurrently active threads. In SIGNAL, this problem reduces to a satisfaction problem. Suppose two requests lock ::=b_1 when c_1 and lock ::=b_2 when c_2 to a lock. Checking non-interference amounts to proving that $c_1 \wedge c_2 = 0$ wrt. the clock constraints inferred by the SIGNAL compiler (which actually does it for free to check that this is compilable as lock ::=b_1 when c_1 default b_2 when c_2.

5 Related work and conclusions

We have presented a novel technology for the integrated modeling, optimization, verification and simulation of embedded system in a functional subset of the real-time JAVA specification (compliant with certifiable software enginneering requirements in avionics) using the multi-clocked synchronous system design platform POLYCHRONY. To our knowledge, PTOLEMY [14] is the only system design platform offering comparable services for component-based design. POLYCHRONY being much more application-domain specific allows for more precise (e.g. clock hierarchization) and aggressive (i.e. global) optimizations (e.g. control) and transformations (e.g. deployment) to be performed. The JAVA to SIGNAL plugin to POLYCHRONY is implemented using SOOT, which offers an effective front-end API and implements efficient sequencial program optimization techniques.

The POLYCHRONY platform provides support for both componenent-based embedded system design, allowing for the capture of existing JAVA components; and refinement-based design, allowing for a seamless upgrade of this components towards deplyoment on a specific target architecture while guarantying compliance to key formal design properties: clock and flow preservation. We have put our polychronous design model and method to work for the refinement of a high-level even-parity checker implementation in JAVA from the early stages of its functional specification to the late stages of its GALS implementation. We have demonstrated the effectiveness of this approach by showing in what respects and at which critical design refinement stages formal verification and validation support was needed, highlighting the benefits of using the tool POLYCHRONY in that design chain.

POLYCHRONY allows to automate required and critical design refinement stages in order to deploy a high-level or prototype system specification, in real-time JAVA, on a resource constrained target architecture, by minimizing the footprint of the required operating system support: scheduling and communication can be minimized to match the requirements of the target architecture. The novelty of integrating POLYCHRONY in a high-level design tool-chain lies in the formal support offered by the former to automate critical and complex design verification and validation stages yielding a correct-by-construction system design and refinement in the latter. Polychronous design allows for an early requirements capture and a compositional and formally checked transformational refinement, automating the most difficult design steps toward implementation using efficient clock resolution and synthesis techniques, implemented in the SIGNAL compiler.

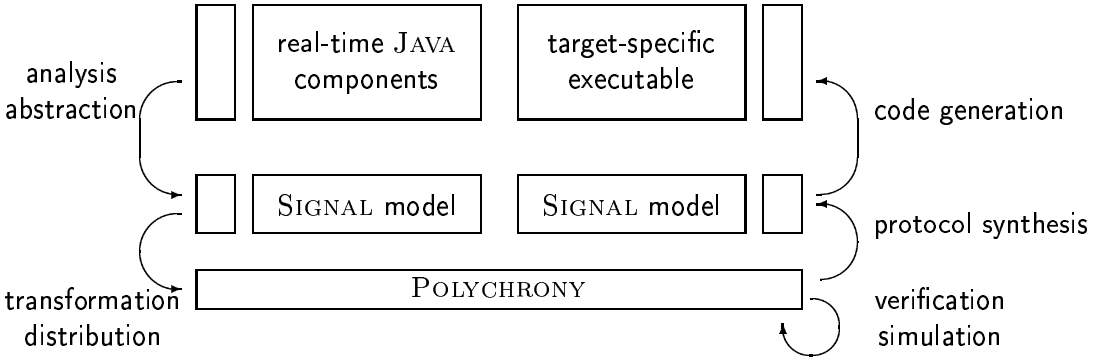


Figure 12: POLYCHRONY as semantics platform for component-based design

In the aim of automating the above process within a versatile component integration platform, the use of POLYCHRONY as a refinement-checking tool provides the required support by using controller synthesis techniques [12]. Whereas model-checking consists of proving a property correct w.r.t. the specification of a system, control synthesis consists of using this property as a control objective and to automatically generate a coercive process that wraps the initial specification so as to guarantee that the objective is an invariant. To this end, we aim at using POLYCHRONY as a semantic platform and SIGNAL as its supportive, intermediate representation for the capture of embedded system component implementations, for instance in real-time JAVA, allowing for a correct by construction component-based design of embedded systems and the systematic synthesis of interface protocols between components.

References

[1] Airlines Electronic Engineering Committee. "Design Guidance for Integrated Modular Avionics". Arinc Report 651-1, November 1997.

[2] Airlines Electronic Engineering Committee. "Avionics Application Software Standard Interface". Arinc Specification 653, January 1997.

[3] Amagbegnon, T. P., Besnard, L., Le Guernic, P. "Implementation of the data-flow synchronous language Signal". In *Conference on Programming Language Design and Implementation*. ACM Press, 1995.

[4] Aubry, P. "Mises en oeuvre distribuées de programmes synchrones" *Thèse de l'Université de Rennes 1*. October 1997.

[5] Benveniste, A., Caspi, P., Le Guernic, P., Marchand, H., Talpin, J.-P., Tripakis, S. "A protocol for loosely time-triggered architectures". In *Embedded Software Conference*. Springer Verlag, October 2002.

[6] Berry, G., Gonthier, G. "The Esterel synchronous programming language: design, semantics, implementation". In *Science of Computer Programming*, v. 19, 1992.

[7] Carloni, L. P., McMillan, K. L., Sangiovanni-Vincentelli, A. L. "Latency-Insensitive Protocols". In *Proceedings of the 11th. International Conference on Computer-Aided Verification*. Lecture notes in computer science v. 1633. Springer Verlag, July 1999.

[8] Gamatié, A., Gautier, T. Modeling of modular avionics architectures using the synchronous language. In *proceedings of the 14th. Euromicro Conference on Real-Time Systems, work-in-progress session*. Ieee Press, 2002. Available as Inria research report n. 4678, December 2002.

[9] Gamatié, A., Gautier, T. The Signal approach to the design of system architectures. In *10th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*. Ieee Press, April 2003.

[10] Lee, E. A., Sangiovanni-Vincentelli, A. "A framework for comparing models of computation". In Ieee *transactions on computer-aided design*, v. 17, n. 12. Ieee Press, December 1998.

[11] Le Guernic, P., Talpin, J.-P., Le Lann, J.-L. Polychrony for system design. In *Journal of Circuits, Systems and Computers. Special Issue on Application Specific Hardware Design, R. Gupta, S. Gupta, S. K. Shukla Eds.* World Scientific, 2002. Available as Inria research report n. 4715, December 2002.

[12] Marchand, H., Bournai, P., Le Borgne, M., Le Guernic, P. Synthesis of Discrete-Event Controllers based on the Signal Environment. In *Discrete Event Dynamic System: Theory and Applications*, v. 10(4), pp. 325–346, 2000.

[13] H. Marchand, E. Rutten, M. Le Borgne, M. Samaan. Formal Verification of Signal programs: Application to a Power Transformer Station Controller. *Science of Computer Programming*, v. 41(1), pp. 85–104, 2001.

[14] E. A. Lee. Overview of the Ptolemy project. *Technical Memorandum UCB/ERL M01/11*. University of California at Berkeley, March 2001.

[15] The Espresso project. `http://www.irisa.fr/espresso`

[16] The Soot project. `http://www.sable.mcgill.ca/soot`

[17] Real-time specification for Java. `http://www.rtj.org`

A Complete JAVA implementation of the EPC (appendix)

Class even

```
import javax.realtime.*;
class even extends PeriodicThread {
    public even () { super(null, new RelativeTime(20,0)); }
    public void run () {
        synchronized (parity.lock) {
            if (parity.start) {
                parity.data = parity.inport;
                parity.istart = true;
                parity.start = false;
            }}
        synchronized (parity.lock) {
            if (parity.idone) {
                parity.outport = parity.ocount & 0x0001;
                parity.done = true;
                parity.idone = false;
            }}}}
```

Class IO

```
class IO extends PeriodicThread {
    public IO() { super(null, new RelativeTime(20,0)); }
    public void run () {
        byte[] buffer = new byte[5];
        synchronized (parity.lock) {
            if (parity.reset) {
                System.out.println("EPC input: ");
                try { System.in.read(buffer, 0, 5); }
                catch (java.io.IOException e) {}
                parity.inport = java.lang.Integer.parseInt
                                (new String(buffer));
                parity.data = parity.inport;
                parity.start = true;
                parity.reset = false;
            }}
        if (parity.done){
            synchronized (parity.lock) {
                System.out.println("EPC output: " + parity.outport);
                parity.reset = true;
            }}}}
```

Class ones

```
class ones extends PeriodicThread {
    public ones () { super(null, new RelativeTime(20,0)); }
    public void run () {
        int data = 0, ocount = 0;
        synchronized (parity.lock) {
            if (parity.istart){
                data = parity.data;
                ocount = 0;
                while (data != 0) {
                    ocount = ocount + (data & 0x0001);
                    data = data >> 1;
                }
                parity.ocount = ocount;
                parity.idone = true;
                parity.istart = false;
            }}}}
```

Class parity

```
class parity {
    public static int inport = 0, outport = 0, data = 0, ocount = 0;
    public static boolean start = false, done = false, istart = false, idone = false, reset = true;
    public static parity lock = new parity();
    public parity(){}
    public static void main (String argv[]) {
        IO Io = new IO ();
        even Even = new even ();
        ones Ones = new ones ();
        Io.start();
        Even.start();
        Ones.start();
    }}
```

B Structure of real-time parameters in the APEX model

```
module TYPES_AND_CONSTANTS_lib =
:
  type APEX_Event_type = struct (
      Comm_ComponentName_type Event_Name;
      EventStateValue_type Event_State;
      WaitRange_type Wait_Processes;
      [MAX_PROCESSES]boolean Wait_Queue;
  );
  type ProcessAttributes_type = struct (
      ProcessName_type Name;
      SystemAddress_type Entry_Point;
      StackSize_type Stack_Size;
      Priority_type Base_Priority;
      SystemTime_type Period;
      SystemTime_type Time_Capacity;
      Deadline_type Deadline;
  );
  type ProcessStatus_type = struct (
      ProcessAttributes_type Attributes;
      Priority_type Current_Priority;
      SystemTime_type Deadline_Time;
      ProcessState_type Process_State;
  );
  type ReturnCode_type = enum (
      NO_ERROR, NO_ACTION,
      UNAVAILABLE, INVALID_PARAM,
      INVALID_CONFIG, TIMED_OUT,
      INVALID_MODE
  );
  type OperatingMode_type = enum (
      IDLE, COLD_START,
      WARM_START, NORMAL
  );
end;
```

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399