

Hard real-time implementation of embedded software in JAVA ^{*} ^{**}

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

INRIA-IRISA, Campus de Beaulieu, 35042 Rennes, France

Abstract. The popular slogan "*write once, run anywhere*" effectively renders the expressive capabilities of the JAVA programming framework for developing, deploying, and reusing target-independent applets. Its generality and simplicity has driven most attention of the compiler technology community to developing just-in-time and runtime compilation techniques, local and compositional optimization algorithms. When it comes to real-time JAVA and to the implementation of embedded software, this approach is however far from satisfactory, especially in hard real-time system design (e.g. airborne systems) where conformance to real-time specifications is critical. We show that synchronous design tools, and particularly the design workbench POLYCHRONY, allow for a complete modeling of embedded software written in a high-level and general purpose programming language such as JAVA. The synchronous approach provides a formal engineering model and methodology, using global transformation and optimization techniques, that allow for a JAVA program written once to be mapped on any distributed target architecture. We present a technique to import a resource constrained, multi-threaded, RT JAVA program, together with its runtime system API, into POLYCHRONY. We put this modeling technique to work by considering a formal, refinement-based, design methodology that allows for a correct by construction remapping of the initial threading architecture of a given JAVA program on either a single-threaded target or a distributed architecture. This technique allows to generate stand-alone (JVM-less) executables and to remap threads onto a given distributed architecture or a prescribed target real-time operating system. As a result, it allows for a complete separation between the virtual threading architecture of the functional-level system design (in JAVA) and its actual, real-time and resource constrained implementation.

1 Introduction

The well-known slogan "*write once, run anywhere*" effectively renders the expressive capabilities of the JAVA programming framework for developing, deploying, and reusing target-independent applets. Its generality and simplicity has driven

^{*} JAVA is a registered trademark of SUN Microsystems

^{**} Work funded by the RNTL project EXPRESSO

most attention of the compiler technology community to developing just-in-time and runtime compilation techniques, local and compositional optimization algorithms. When it comes to real-time JAVA (RT JAVA) and to the implementation of embedded software, this approach is however far from satisfactory, especially in hard real-time system design (e.g. airborne systems) where conformance to real-time specifications is critical.

We show that synchronous design tools, and particularly the design workbench POLYCHRONY¹, allow for a complete modeling of embedded software written in a high-level and general purpose programming language such as JAVA. The synchronous approach provides a formal engineering model and methodology, using global transformation and optimization techniques, that allow for a JAVA program written once to be mapped on any distributed target architecture.

We present a technique to import a resource constrained, multi-threaded, RT JAVA program, together with its runtime system API, into POLYCHRONY. We put this modeling technique to work by considering a formal, refinement-based, design methodology that allows for a correct by construction remapping of the initial threading architecture of a given JAVA program on either a single-threaded target or a distributed architecture. This technique allows to generate stand-alone (JVM-less) executables and to remap threads onto a given distributed architecture or a prescribed target real-time operating system. As a result, it allows for a complete separation between the virtual threading architecture of the functional-level system design (in JAVA) and its actual, real-time and resource constrained implementation.

We illustrate this technique by considering a simple yet representative case study: an even-parity checker (EPC). We show how correctness by construction is enforced from the description of its functional architecture (in RT JAVA) to its mapping onto a given target architecture. This case study demonstrates the ability of the POLYCHRONY workbench to automatically perform otherwise difficult engineering tasks such as thread remapping and target-specific deployment, starting from a given, high-level, RT JAVA design.

Overview In section 2 we present the considered functional profile of RT JAVA. After a brief introduction to our modeling platform POLYCHRONY in section 3, section 4 describes the architecture of our modeling tool. In section 5 we finally describe how to take advantage of the formal techniques of POLYCHRONY in order to address critical issues in hard real-time system design.

2 A hard RT JAVA profile

The RT JAVA profile (i.e. the API of the JVM extension) considered in the present article has been implemented in the context of the EXPRESSO project² and inspired by the Ravenscar-Java "high-integrity profile" (HIP) for RT JAVA [12]

¹ <http://www.irisa.fr/espresso/Polychrony>

² <http://www.irisa.fr/rntl-expresso>

(there are other specifications for RT JAVA such as [4]). It consists of a subset of the RT JAVA specification aimed at meeting non-functional requirements of airborne systems. Table 1 details the most significant features of this profile (some constructs like memory management are not considered in this article). The EXPRESSO packages provide extensions to thread management classes allowing for the simulation of real-time threads and event handlers. The class `AsyncEvent` and its sub-classes define data-structures to encapsulate hardware interrupts and events to be recycled within the real-time JVM by appropriate handlers (class `AsyncEventHandler` and sub-classes) according to the pace of the real-time kernel. Processing in nominal mode within a real-time application is performed using real-time and periodic threads (classes `RealTimeThread` and `PeriodicThread`). These allow to schedule the execution of a sequential piece of code according to real-time constraints (period, deadline, duration, priority) and can be set using shared data-structures defined in the `SchedulingParameters` class.

Table 1. Excerpt of the EXPRESSO package class hierarchy

<pre> .../... class SchedulingParameters class AsyncEvent class SporadicEvent class SporadicInterrupt class AsyncEventHandler (implements Schedulable) class BoundAsyncEventHandler </pre>	<pre> class RealtimeThread (implements Schedulable) class Initializer class NoHeapRealtimeThread (implements Schedulable) class PeriodicThread class MonitorControl .../... </pre>
--	--

Airborne software requirements The HIP profile for RT JAVA further describes programming guidelines for the use of this profile to meet structural and functional requirements imposed by certification authorities. The syntactic simplicity of these requirements make them at the same time easy to translate into programming guidelines by users, easy to implement as syntactic checks with the help of a modeling tool, and easy to analyze using rate-monotonic analysis techniques.

- A program consists of a fixed number of threads.
- Thread and memory allocation is performed during service startup.
- No dynamic memory management during operational service.
- Threads have access to the scope of the program.
- Threads are either periodic or sporadic.
- Threads use synchronization to avoid priority inversion.

In the present study towards modeling RT JAVA within the synchronous multi-clocked design workbench POLYCHRONY, we center our focus on this very subset of JAVA and demonstrate that its functionalities fit within our model of computation.

An intermediate format To allow for the translation of a RT JAVA program into POLYCHRONY starting from either its source code or its bytecode, we use the tool SOOT³ as JAVA compilation front-end to pre-process classes and obtain an optimized JIMPLE intermediate representation. JIMPLE is a very handy format to process JAVA classes. It consists of explicitly typed, stack-less, 3-address statements (grammars *stm* and *rtn*) that manipulate either immediates or references (grammars *i* and *v*).

(immediate)	$i ::= l \mid c$	(register or constant)
(operator)	$\star ::= + \mid - \mid \dots$	(native plus, minus, etc)
(variable)	$v ::= i[i] \mid i.[x] \mid x \mid l$	(array , class field or local)
(reference)	$r ::= \text{caughtexception}$	(current exception)
	$\text{parameter } c$	(method parameter)
	this	(self)
(declaration)	$dec ::= v = i \text{ instanceof } t$	(instantiation)
	$v = \text{new } t[i]$	(memory allocation)
	$t \ x$	(type declaration)
(program)	$run ::= blk \mid run; run$	(sequence of blocks)

Fig. 1. A grammar of JIMPLE (programs)

In the JIMPLE grammar (Figure 1), a local variable is noted l , a constant c , a type t , a program label L , and class field name x . 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*. A declaration (grammar *dec*) may not occur in the *run* method of a thread or in an event handler as it might dynamically allocate memory. Hence, all declarations are assumed to be present in the *main* initialization class of the program, or in the *init* method of thread classes (executed once at system start).

(block)	$blk ::= L : stm^*; rtn$	(sequence of <i>stms</i>)
(statement)	$stm ::= v = i \star i$	(native operation)
	$v = \text{invoke } i (i^*)$	(method invocation)
	$l := [v \mid @r]$	(local variable)
(return)	$rtn ::= [\text{enter} \mid \text{exit}] \text{monitor } i$	(locks)
	$\text{goto } L$	(goto)
	$\text{if } i \text{ then } L$	(test)
	return	(return)
	$\text{throw } i$	(throw exception)
	$\text{catch } t \text{ from } L \text{ to } L \text{ using } L$	(catch exception)

Fig. 2. A grammar of JIMPLE (statements)

As a result, the SOOT toolbox provides an appropriate front-end to resolve high-level object-oriented features and perform specific optimizations, allowing

³ <http://www.sable.mcgill.ca/soot>

us to focus on the translation of the JIMPLE imperative notation into the multi-clocked data-flow design language SIGNAL, presented next. JIMPLE produces explicitly typed and initialized declarations as well as explicit locks in the presence of exceptions (monitors are released before exceptions are raised).

3 POLYCHRONY for embedded system design

The POLYCHRONY workbench implements a multi-clocked synchronous model of computation (the polychronous MOC [9]) to model control-intensive embedded software using the SIGNAL language and to support a formal, refinement-based, design methodology [13] with companion decision procedures to validate key design steps using precise formal design properties (e.g. controllability of a component by its environment or invariance of a design refinement under flow-equivalence) and/or automatic design transformations and refinement algorithms (control hierarchization, protocol synthesis).

3.1 An introduction to SIGNAL

SIGNAL belongs to the family of synchronous languages such as ESTEREL and LUSTRE. A SIGNAL *process* P consists of simultaneous equations over *signals*. A signal $x \in \mathcal{X}$ describes an infinite flow of values $v \in \mathcal{V}$. We write \mathbf{x} for a sequence or tuple (x_1, \dots, x_n) of signals. An equation $\mathbf{x} := f\mathbf{y}$ denotes a relation between a sequence of input signals \mathbf{y} and a sequence of output signals \mathbf{x} by a function or combinator f . SIGNAL requires three primitive operators: the equation $x := y$ init v initially defines x by v and then by the previous value of y in time, the equation $x := y$ when z defines x by y when z is true and the equation $x := y$ default z defines x by y when y is present and by z otherwise. The synchronous composition $P \mid Q$ of two processes P and Q consists of the simultaneous solution of the system of equations P and Q .

$$P ::= \mathbf{x} := f\mathbf{y} \mid (P \mid Q) \mid P \text{ where } \mathbf{x}$$

A first example. As an illustration we consider the definition of a simple counting process `Count`, below. It accepts an input signal `rst` and delivers the integer output signal `val`. A local variable `cnt`, initialized to 0, stores the previous value of `val` (equation `cnt := val` 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 controlled by the clock of its output `val` which differs from that of its input `rst`. We write `process Count = (? event rst ! integer val)` for the declaration of a process named `Count` of input `rst` and of output `val`.

<code>process Count = (? event rst ! integer val)</code>	<i>time</i>	<code>t₁</code>	<code>t₂</code>	<code>t₃</code>	<code>t₄</code>	<code>t₅</code>	<code>t₆</code>	<code>t₇</code>	<code>t₈</code>	<code>t₉</code>	<code>t₁₀</code>
<code>(cnt := val</code> init 0	<code>rst</code>	true						true			
<code> val := (0 when rst) default (cnt + 1)</code>	<code>val</code>	1	0	1	2	3	4	0	1	2	3
<code>) where integer cnt end;</code>	<code>cnt</code>	0	1	0	1	2	3	4	0	1	2

Clocks and causality relations. In SIGNAL, sequential code generation (to, e.g., ANSI C, JAVA, VHDL) is a design stage performed on a given model (e.g. the modulo 3 counter, figure 3) subsequently to an analysis of synchronization relations (e.g. signals i , s and o are synchronous, written $o \hat{=} s \hat{=} i$, figure 3) and scheduling relations (e.g. c and o cannot happen before s when i is present, written $s \rightarrow^i c$ and $s \rightarrow^i o$, figure 3) are inferred. This relational information is used to construct a global and canonical control flow graph (described next). This transformation allows, for instance, to synthesize a single automaton from the model of multiple threads. Conversely, it allow to synthesize synchronization protocols to correctly map these threads on a distributed architecture.

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

Fig. 3. From synchronous multi-clocked equations to sequential C code

The hierarchization of the control flow graph (Figure 4, from [3]) is the key transformation performed by POLYCHRONY on a given SIGNAL specification. Given an inferred clock relation (e.g. $h_3 = h_1 \text{ op } h_2$, below), it allows to optimally place clocks (e.g. h_3 , below) in the control-flow tree by determining their least upper bound (e.g. h , s.t. $h > h_1$ and $h > h_2$, below). Each clock (e.g. c for $(s == 2)$, above) is the trigger of a set of actions that are scheduled in sequence by respecting the inferred scheduling relations (e.g. $s \rightarrow^i o$, above).

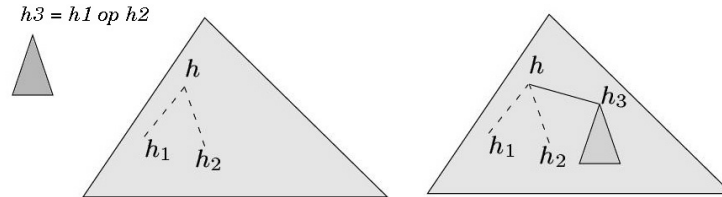


Fig. 4. Hierarchization of a control-flow graph for scheduler generation.

3.2 An introduction to the APEX-ARINC RTOS model of POLYCHRONY

The APEX⁴ interface, defined in the avionics standard ARINC [2], provides avionics application software with basic services needed to access operating system 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 levels of criticality) can be hosted on a single, shared computer system. Of course, an essential issue is to ensure safe allocation of shared computer resources in order to prevent fault propagation from

⁴ APEX (**A**pplication **E**xecutive) is a real-time operating system standard API for avionics applications

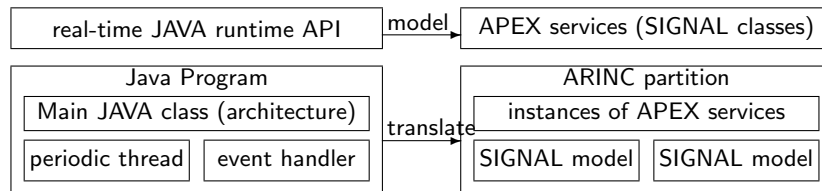
one hosted application to another. This is addressed through functional partitioning of the applications w.r.t. available time and memory resources. The allocation unit that results from this decomposition is the *partition* (see Figure 7 in the EPC example). A partition is composed of *processes* which 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 needed 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 communication, synchronization, and services for the management of processes and partitions. Such services have been modeled in POLYCHRONY [6] and its commercial implementation (RT-builder, by TNI-VALIOSYS, <http://www.tni-valiosys.com>) is used at Hispano-Suiza and Airbus Industries. We use these services to model and implement hard RT JAVA applications.

4 A hard RT JAVA plugin for POLYCHRONY

We introduce our method to translate and model multi-threaded real-time JAVA programs in POLYCHRONY. Starting from a given, multi-threaded, RT JAVA design, POLYCHRONY allows to remap the functional thread architecture and to retarget the design onto a different target real-time operating system. Furthermore it can generate stand-alone (JVM-less) executables. As a result, it allows for a complete separation between the virtual threading architecture of the functional-level system design and its actual, real-time implementation.

4.1 Modeling the RT JAVA virtual machine using POLYCHRONY.

Modeling a RT JAVA application in POLYCHRONY requires to distinguish between the architecture of the application and its periodic threads and event handlers. The architecture (i.e. shared data-structures and thread parameters) can be obtained by scanning the main (initialization) class of the JAVA program as well as the init methods of thread classes.



The periodic threads and event handlers are described in the run methods of the thread classes. The init methods make use of operating system support (the RT JAVA API) that is modeled using the APEX services library of POLYCHRONY.

The architecture of the entire framework is formed by the APEX services of POLYCHRONY that model the RT JAVA API and correspond to the virtual machine. The structure of the actual JAVA program, that describes the functional threading and memory architecture, is translated to instances of APEX services and ARINC partitions. JAVA threads and event handlers are translated into SIGNAL processes.

4.2 Modeling RT JAVA threads using POLYCHRONY

To model RT JAVA threads we consider the JIMPLE intermediate format. In this format the model of a JAVA class is defined by the translation function $\llbracket run \rrbracket_E = \langle p \rangle$ and $\llbracket blk \rrbracket_E^C = \langle p \rangle$ which takes the code of an input method *run* as input, along 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* raised 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 (Figure 7).

A real-time periodic thread (or event handler) is viewed as a sequence of critical sections that receive control from the partition-level scheduler via a clock tick, which triggers the execution of the thread, and a variable `next_block`, which directs control to the very block to execute in the thread. The type of the `next_block` variable is the enumeration of the block label names *L*, which are referenced to as *#L* in SIGNAL.

A *section* or block *blk* 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. A *statement stm* takes three forms:

*The definition $l := r$ of a local variable *l*.* The use of local variables in JIMPLE code facilitates 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.

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

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

*Lock monitoring is modeled by inlined SIGNAL processes (e.g. `entermonitor` and `exitmonitor`). *Control*, via `goto` or `if`, consists in the activation of the clock that corresponds to the target block label. The handling of exceptions does not depart*

from this scheme: an *exceptional* control-flow branch is produced by SOOT to handle a `throw` in a way similar as a `goto`, by associating the corresponding `catch` to a label. In the particular case of the `return` statement, translation is performed by installing the corresponding pattern of the APEX protocol at partition level (see Figure 7). Variables v and references r are translated as is $(i.[x])$ as $i.x$ (a datum) or $i.x$ (a method), parameter c stays as is, `this` is removed, etc.).

$$\begin{aligned}
& \llbracket blk; run \rrbracket_E = \langle p \mid q \rangle \text{ where } \llbracket blk \rrbracket_E = \langle p \rangle \text{ and } \llbracket run \rrbracket_E = \langle q \rangle \\
& \llbracket L : stm_{1\dots n}; rtn \rrbracket_E^C = \langle C_E(L) ::= \text{when next_block}\$1 = \#L \text{ when tick} \mid p_1 \mid \dots \mid p_n \mid p \rangle \\
& \quad \text{where for } 0 < i \leq n, \llbracket stm_i \rrbracket_E^C = \langle p_i \rangle \text{ and } \llbracket rtn \rrbracket_E^C = \langle p \rangle \\
& \llbracket l := v \rrbracket_E^C = \langle l ::= v\$1 \text{ when } C \rangle \\
& \llbracket l := @r \rrbracket_E^C = \langle l ::= r \text{ when } C \rangle \\
& \llbracket v = \text{invoke } i(i^*) \rrbracket_E^C = \langle v ::= \mathcal{R}_E(i)((i^*) \text{ when } C) \rangle \\
& \llbracket v = i \star j \rrbracket_E^C = \langle v ::= (i \star j) \text{ when } C \rangle \\
& \llbracket \text{entermonitor } i \rrbracket_E^C = \langle \text{entermonitor}\{S, \mathcal{R}_E(i)\}(\text{when } C) \rangle \\
& \llbracket \text{exitmonitor } i \rrbracket_E^C = \langle \text{exitmonitor}\{S, \mathcal{R}_E(i)\}(\text{when } C) \rangle \\
& \llbracket \text{goto } L \rrbracket_E^C = \langle \text{if } (L \notin \text{pred}^*(\mathcal{G}_E)(L)) \text{ then } \langle C_E(L) ::= \text{when } C \rangle \\
& \quad \quad \quad \text{else } \langle \text{next_block} ::= \#L \text{ when } C \rangle \\
& \llbracket \text{if } i \text{ then } L \rrbracket_E^C = \langle \text{if } (L \notin \text{pred}^*(\mathcal{G}_E)(L)) \text{ then } \langle C_E(L) ::= \text{when } i \text{ when } C \rangle \\
& \quad \quad \quad \text{else } \langle \text{next_block} ::= \#L \text{ when } i \text{ when } C \rangle \\
& \llbracket \text{throw } i \rrbracket_E^C = \langle C_E(L_2) ::= \text{when } C \mid C_E(L_1) ::= C_E(L_2) \text{ when } C \rangle \\
& \quad \text{where } C = C_E(L) \text{ and } \mathcal{X}_E(\mathcal{R}_E(i), L) = (L_1, L_2) \\
& \llbracket \text{return} \rrbracket_E^C = \langle \text{next_block} ::= \text{main}(\mathcal{G}_E) \text{ when } C \rangle
\end{aligned}$$

Fig. 5. Import of a RT JAVA thread from JIMPLE to SIGNAL

4.3 A case study

To illustrate our modeling principles and outline its application to the implementation of a formal refinement-based design methodology, we study the refinement of the functional architecture of an even-parity checker (EPC) towards its distributed implementation. This case study - even if it is small compared to what could be handled by the tool - demonstrates the capability of the POLYCHRONY workbench to automatically perform otherwise challenging engineering tasks such as thread remapping, generation of stand-alone executables, or target-specific deployment, starting from a given, high-level, RT JAVA design.

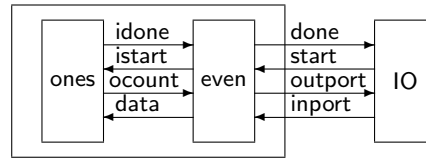


Fig. 6. Model of an even-parity checker in POLYCHRONY

The EPC consists of three functional units shown in Figure 6: an IO interface process, a master `even` parity check process, and a slave `ones` bit-counting process. The IO process will give a `start` signal to the `even` process and will then wait for the signal `done` to read the result. On `start`, `even` will read the input data, pass it to the process `ones` and notify it with the `istart` signal. `Ones` counts the number

of bits of the input data that are true and notifies even about the completion. Even finally checks if the result is an even number and notifies IO.

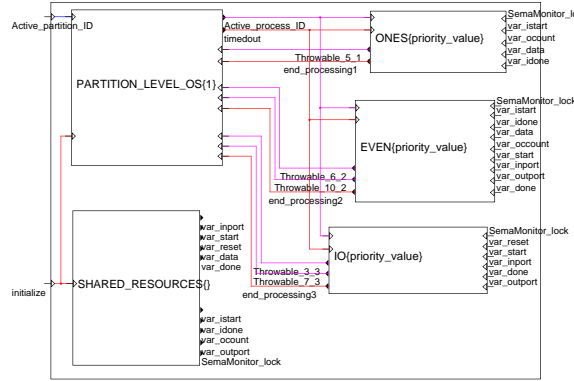


Fig. 7. Architecture of the ARINC partition for the EPC.

Example (thread ones). Having grasped the structure of the whole system and the architecture (Figure 7) of our case study, we have a closer look at the translation of one specific thread, the ones thread. The concurrency model of RT 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 (Figure 8, left) determines the parity of an input data. Upon receipt of the start notification, ones shifts data until it becomes 0 and the internal count is assigned to the output count and done notified. The thread even notifies ones to start processing data and waits until done is notified to read the final count and checks whether it is an even number.

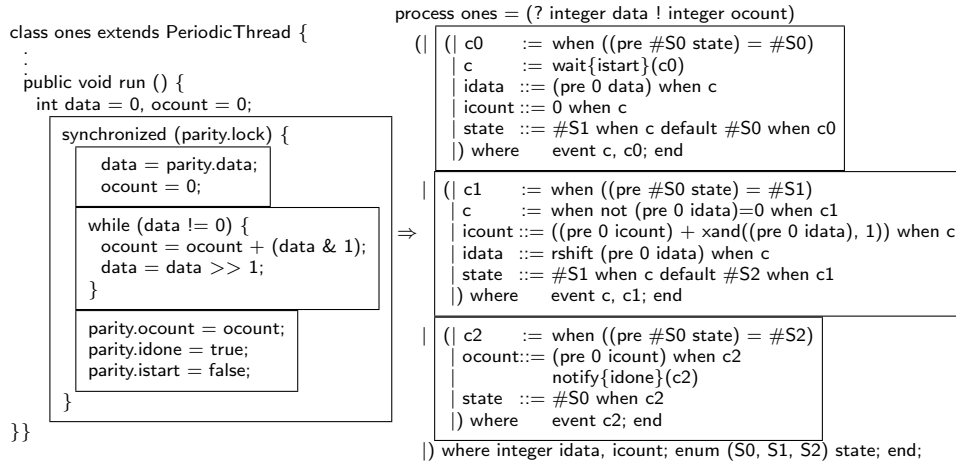


Fig. 8. Translation of the ones threads in POLYCHRONY

The SIGNAL model of thread ones (figure 8) consists of one critical section, delimited by a pair of monitor statements. The process is activated when it ob-

tains the lock on `istart`. Then, at its own rate (now conditioned by the clock `c1`), it determines the result. When it is finished, it sends the notification. Native operators, e.g. the unsigned operators `>>` and `&`, or separately compiled functions can be imported into the model as external functions with a behavioral type specification (the `spec` declaration).

```
function rshift = (? i1 ! i2) spec (| i1 ^ = i2 | i1 → i2 |)
    pragmas JAVA_CODE "i2 = i1 >> 1" end pragmas;
function xand = (? i1, i2 ! i3) spec (| i1 ^ = i2 ^ = i3 | i1 → i3 | i2 → i3 |)
    pragmas JAVA_CODE "i3 = i1 & i2" end pragmas;
```

Example (architecture of the EPC main). To illustrate how a RT 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` (Figure 9). 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 handlers 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.

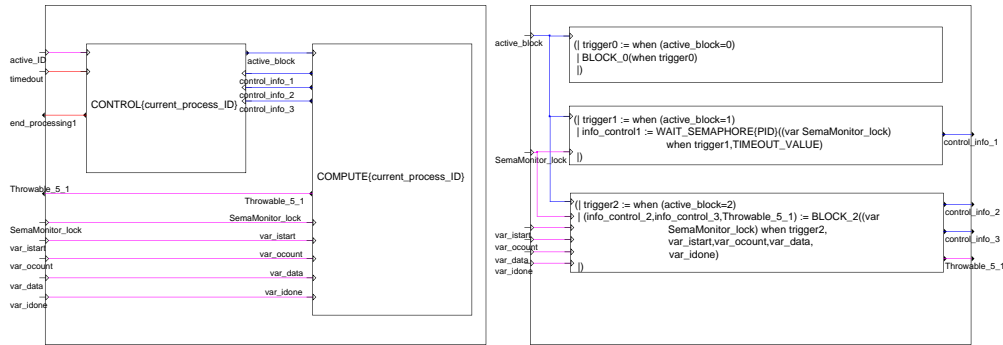


Fig. 9. Control and computation nodes of the ARINC model for thread ones.

These data are used to instantiate the generic APEX service models of POLY-CHRONY (figure 9) and finalize the model of the application architecture. This yields the structure depicted in Figure 9 for the `ones` thread: a control process is connected to the partition-level scheduler and a computation process.

5 Checking the correctness of system design refinements

In this section, we demonstrate the use of our modeling tool to check the refinement of the even-parity checker (EPC) from its initial specification towards its distributed implementation. This is done by the insertion of communication 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 us to consider high-level system component descriptions and to refine them in a semantic-preserving manner into a GALS implementations (globally asynchronous locally synchronous, aka. Kahn network).

Architecture design refinement. To model the physical distribution of the threads `ones` and `even` and in order to allow them to communicate asynchronously via a channel structure, we install a double handshake protocol between them. The installation of this channel incurs a desynchronization between the two processes, hence a potential change of behavior. The model of the `send` and `recv` methods in SIGNAL is obtained from a message sequence specification (Figure 10, left). The `ready` and `ack` flags stand for state variables declared in the lexical scope of `send` and `recv` in the module that defines the protocol; `eReady` and `eAck` stand for events. Sender and receiver use a simplified wait/notify mechanism similar to that of asynchronous event handlers. In [13], we show that the validation of a protocol insertion such as the EPC model upgrade with a double-handshake protocol amounts to checking that the initial and upgraded designs have equivalent flows, which is amenable to model checking by proving that both design outputs always return the same sequence of values.

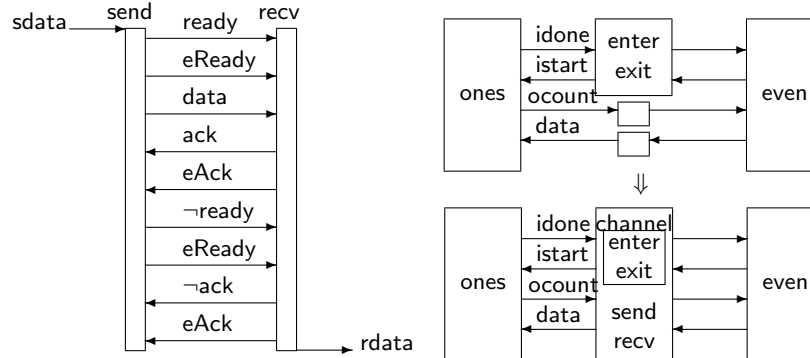


Fig. 10. Refinement of the EPC architecture with the double-handshake protocol

Remapping threads onto a target. 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, the SIGNAL compiler can be used to merge the behaviors `IO` and `even` and combine their control-flow using the clock hierarchization algorithm.

Then, checking the merge of the threads IO and even correct w. r. t. the initial specification amounts to checking that it is deterministic. As demonstrated in [9], this is done by checking that the clock of its output can be determined from that of its input `data` and from the master simulation tick and that checking that the frequency of the output is lower than that of both inputs `data` and `tick`.

Table 2. Execution times of multi-threaded applications [ms]

Case study	JCC	TURBOJ	POLYCHRONY
I	145	139	50
II	1428	1375	450
III	14312	13677	4540

In [5], experiments made in the context of the RNTL-EXPRESSO project are reported, showing that for several case-studies (multi-threaded RT JAVA programs consisting of five to ten threads), the speedup obtained by automatically remapping all threads into a stand-alone (JVM-less) and serialized executable was in average 300% compared to a commercial JAVA compiler implementing the RT JAVA specification (TURBOJ) and to the standard JAVA compiler (JCC).

Note that the construction of a single automaton from multiple JAVA threads may not be always possible: the corresponding SIGNAL model may not necessarily exhibit a single control-flow tree after hierarchization (figure 4). However, it may still be possible to repartition the model into a smaller set of threads (at most equal to the initial one) having a hierarchizable control-flow tree.

Proving global design invariants. In addition to methodology-specific verification issues addressed in the present article, the model checker of SIGNAL allows to prove more general properties of specification requirements: reachability, attractivity, and invariance (see [10] for details). Example 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 [7], in the context of the modeling of APEX avionics application in SIGNAL and the companion design methodologies. Another common requirement is non-interference: e.g. a lock is never requested from two concurrently active threads. In SIGNAL, this problem reduces to a satisfaction problem. Suppose two requests `lock ::= b1 when c1` and `lock ::= b2 when c2` to a lock. Checking non-interference amounts to proving that $c_1 \wedge c_2 = 0$ w. r. t. clock constraints. These constraints are inferred by the SIGNAL compiler for a specific hard real-time implementation.

6 Related work and conclusions

The present work is based on previous results in the POLYCHRONY project. It uses the polychronous model of computation [9], the model of the APEX real-time operating system standard [6], and a formal refinement-based design

methodology [13]. PTOLEMY [11] and ROSETTA [8] are approaches that partly aim at a similar goal. They examine system implementations using different computation models. The synchronous model, which we consider, is only one of these, but it allows the treatment of more general, multi-clocked systems. We presented a new engineering technique allowing for the integrated modeling, optimization, verification, and simulation of embedded systems in a functional subset of the RT JAVA specification, which is compliant with certifiable software engineering requirements in avionics. This platform-based design using the multi-clocked synchronous framework POLYCHRONY allows to perform precise and aggressive optimizations and transformations, that would be hard, yet impossible to achieve using common techniques.

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. BOLLELA, G. ET AL "The real-time specification for JAVA". Addison-Wesley, 2000.
5. D. COSTARD. Evaluation d'une chaîne de compilation JAVA temps-réel. *Rapport de stage de fin d'étude*, ESIEE, Juin 2003.
6. 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.
7. 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.
8. Perry, A., Kong, C.. Rosetta: Semantic support for model-centered systems-level design. In *IEEE Computer*, 34(11):6470, November 2001.
9. 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.
10. MARCHAND, H., RUTTEN, E., LE BORGNE, M., SAMAAAN, M. Formal Verification of SIGNAL programs: Application to a Power Transformer Station Controller. *Science of Computer Programming*, v. 41(1), pp. 85–104, 2001.
11. J.T. BUCK, S. HA, E.A. LEE AND D.G. MESSERSCHMITT. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. In *International Journal of Computer Simulation, special issue on "Simulation Software Development"*, v. 4, pp. 155-182. Ablex, April 1994.
12. KWON, J., WELLINGS, A.J., KING, S. A High Integrity Profile for Real-time Java. *Proceedings of the Joint ACM Java Grande Conference*. ACM press, 2002.
13. TALPIN, J.-P., LE GUERNIC, P., SHUKLA, S. K., GUPTA, R., DOUCET, F.. Polychrony for formal refinement-checking in a system-level design methodology. In *Application of Concurrency to System Design*. IEEE Press, June 2003.