A methodology and tool support for generating scheduled native code for real-time Java applications *

Christos Kloukinas Chaker Nakhli Sergio Yovine

VERIMAG
Centre Equation, 2 Ave. Vignate, 38610 Gières, France
{Christos.Kloukinas,Chaker.Nakhli,Sergio.Yovine}@imag.fr

1 Introduction

Current trends in industry are leading towards the use of Java [5] as a programming language for implementing embedded and real-time applications. From the software engineering perspective, the Java environment is indeed a very attractive development framework. Object-oriented programming provides encapsulation of abstractions into objects that communicate through clearly defined interfaces. Dynamic loading eases the maintenance and improvement of complex applications with evolving requirements and functionality. Besides, Java provides built-in support for multi-threading.

However, the semantics of Java do not guarantee a predictable run-time behavior, which is an essential issue in embedded real-time software. To overcome this problem, work has been done to extend the language and the platform to accommodate to the requirements of real-time systems by focusing on current practices. Among such work, we should mention the Real-Time Specification for Java [11], and the Real-Time Core Extension for the Java Platform [12], that provide support for real-time programming (timers, clocks, handlers, priorities, . . .). Still, these extensions leave some important issues unspecified, like the scheduling algorithm to be used, allowing an implementation to resolve them at will.

In order to obtain more precise semantics, domain-specific profiles have also been defined, such as the Ravenscar-Java [8] high-integrity profile for safety-critical systems. This profile settles an execution model based on a two/three-phase program execution, comprising an initialization phase and a mission phase (possibly followed by a termination phase), and multi-threading semantics relying on fixed priority preemptive scheduling and priority ceiling inheritance. Though designed to ease analysis and programming, this profile still has some drawbacks. For instance, it does not directly support threads which synchronize and communicate using the `wait` and `notify` methods of Java. Besides, the underlying schedulability analysis is pessimistic by nature and not well adapted for systems with heterogeneous tasks and constraints.

The other important issue is performance. Though there are efforts to produce efficient implementations of Java Virtual Machines (*e.g.*, [1,15]), the slowdown due to the VM remains an argument against adopting Java for real-time applications. Real-time systems can afford neither the overhead nor the non-determinism of using a Just-In-Time (JIT) compiler (*e.g.*, [3,14]). An alternative approach consists in using an Ahead-

* Partially supported by the RNTL project Expresso (http://www.irisa.fr/rntl-expresso).

of-Time (AOT) compiler (*e.g.*, [10,17]) to generate executable code for a run-time system and to provide a native implementation of the real-time primitives. A major advantage of AOT compilation is that it allows performing sophisticated analysis techniques to produce highly optimized code.

In this paper we present an approach that takes into account the demands of both precise semantics and performance. Our work is based on the two/three-phase execution model and API of the Expresso High-Integrity Profile [4], which itself inherits concepts from the Ravenscar-Java profile and the RTSJ API. However, the semantics proposed by the profile do not fit the needs of applications that would demand alternative scheduling and synchronization paradigms, and handling quality of service requirements. To accommodate to such demands, our approach focuses primarily on the application.

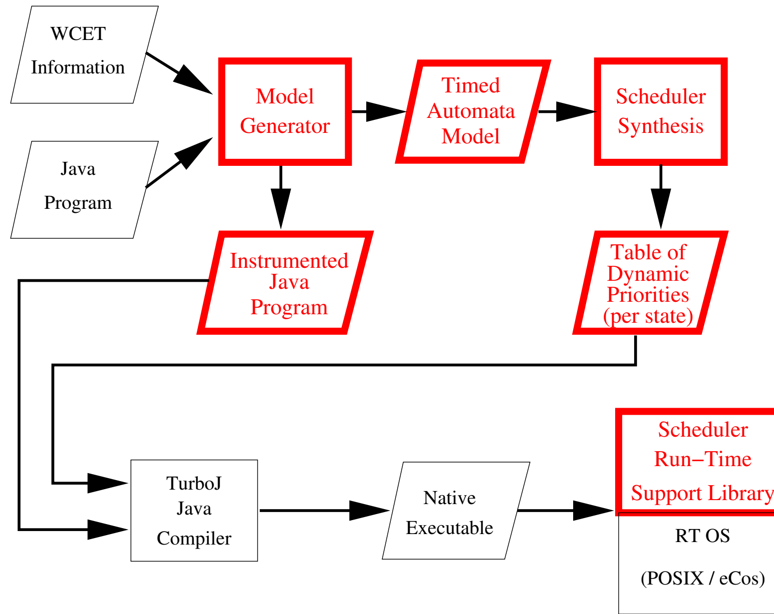

Fig. 1. Code analysis & generation chain

Following [13], we first extract a formal model of the behavior of the application program as an extended automaton (see Fig. 1 & section 2). Then, we synthesize an application-dependent scheduler (see sections 3–4) which is *safe* (*i.e.*, it is deadlock free and meets all timing constraints) and *QoS extendible* (*i.e.*, it can be extended to handle QoS requirements, such as reducing response jitter, power consumption or context switches). This synthesized scheduler is meant to be used with an appropriate native run-time support we have developed, which itself uses the underlying *R-T OS*'s primitives (see section 5). Our scheduler also needs an instrumented version of the original Java code (also produced by our model extracting tool), so as to be able to follow

the changes of thread states (*i.e.*, the instrumentation implements an abstract program counter). The test-bed we have developed has been integrated together with the Expresso High-Integrity API and the TurboJ [17] Java-to-native compiler.

This framework provides a complete analysis and compilation chain for embedded real-time systems based on Java, allowing one to substitute RMA/EDF & PCP with a scheduler which is still safe but not as pessimistic. In this article, we describe the model generator, the scheduler architecture and synthesis methodology, and the prototype test-bed implemented using POSIX [6] primitives.

We will illustrate our approach throughout the paper with a case study inspired by the robotic arm system described in [9] (see Fig. 2). The arm is programmed to take objects from a conveyor belt, to store them in a buffer shelf, and to put them into a basket. The arm is controlled by threads running on a single processor. The `TrajectoryControl` thread reads commands from a shared buffer and issues set-points to the low-level arm controller `Controler`. If there are no commands to process, it holds, otherwise reads sensor values and computes the new set-point. Its execution time is between 5ms and 6ms. The `Lifter` thread is activated periodically every 40ms. Its role is to command the arm to pick objects from the conveyor belt. Upon termination, it issues a command to the `TrajectoryControl` and activates the `Putter`. Its execution time is between 4ms to 8ms. The `Putter` thread sends commands to take the object from the buffer shelf and put it into the basket. Its execution time is between 4ms to 8ms. The `SensorReader` thread reads sensors every 24ms. Its execution time is 1ms. The results of the sensors are used by `TrajectoryControl`. `Controler` is a periodic thread with period 16ms. Notice in Fig. 2 how, according to the Expresso HIP-API, **waitForPeriod** returns a boolean value which is **false** if the thread misses its period. In such a case, the application ends the mission phase and goes into a termination phase which is omitted here. In this paper we only consider the problem of synthesizing a scheduler for the mission phase.

The paper is organized as follows. Section 2 presents the technique we use to generate models from Java source code. Section 3 explains the scheduler architecture and execution semantics, while section 4 describes our methodology for synthesizing an application-dependent scheduler. Section 5 discusses our test-bed implementation.

2 Model Generation

We consider real-time Java applications made up of a fixed number of threads that synchronize and communicate through a fixed set of global shared objects. There is a distinguished thread, namely `Main`, which is the first thread to wake up at application startup and the unique entry-point of the application. We assume that `Main` is programmed according to the Expresso HIP, that is, all the shared objects and threads are created during the initialization phase.

The model of a Java program is a transition system which abstracts program actions and states. Each state in the model is an abstraction of a program state at run-time. Transitions are associated with source code and capture the change that its execution makes to the program state. More formally, let $\Theta = \{th_i\}$ be a finite set of threads, $\Omega = \{O_i\}$ be a finite set of shared objects, and A be a set of labels. Labels may correspond to

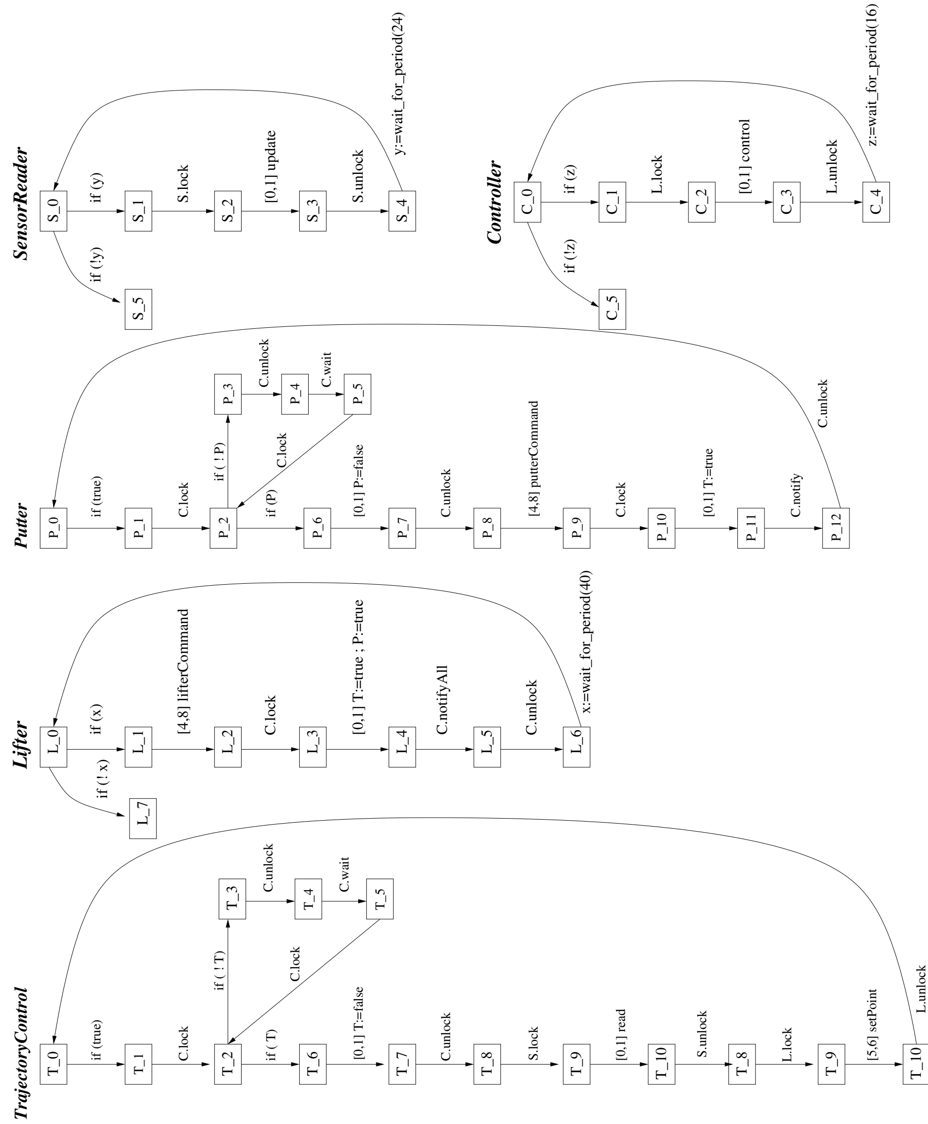

Fig. 2. Robotic arm system architecture

Intervals preceding computations give their minimum & maximum execution duration. Transitions $T_9 \rightarrow T_10$ (setPoint), $L_1 \rightarrow L_2$, $P_8 \rightarrow P_9$, $S_2 \rightarrow S_3$ and $C_2 \rightarrow C_3$ correspond to code which has been sliced away.

large blocks of source code or to specific statements, such as: **lock** (corresponding to the Java-bytecode **monitorEnter**), **unlock** (**monitorExit**), **wait**, **waitTimed** (the Java **wait** method with a timeout parameter), and **waitForPeriod** (the method of the class `PeriodicThread` in the Expresso profile, which blocks a thread until its next period). The model is a tuple $P = (S, A, T)$ where: S is a finite set of states, and $T \subseteq S \times A \times S$ is a transition relation. We define $TH : T \longrightarrow \Theta$ to be the function mapping each transition to the corresponding executing thread.

For each thread th we define $\Sigma_{th} : S \longrightarrow 2^{\Omega} \times 2^{\Omega}$, where $\Sigma_{th}(s) = (\Sigma^+, \Sigma^-)$ is called the *synchronization context* of th at the state s. Σ^+ contains the set of objects which are locked by th, when th is at state s. Σ^- is either the empty set or a singleton containing the object that cannot be synchronized by th at s (and thus cannot be added to Σ^+) which corresponds to the lock released when **wait** or **waitTimed** are invoked on that object. This is done to keep the synchronization context consistent during the model extraction process. Let $\Sigma = (\Sigma^+, \Sigma^-)$ be a synchronization context, and $\omega \subset \Omega$ a set of global objects. We define:

$$\Sigma \; Add \; \omega = (\Sigma^+ \cup (\omega \setminus \Sigma^-), \Sigma^-)$$
$$\Sigma \; Remove \; \omega = (\Sigma^+ \setminus \omega, \Sigma^- \cup \omega)$$

The Add operation appends objects to the synchronization context by adding them to the Σ^+ set, as long as these objects are not in the Σ^- set. The $Remove$ operation removes objects from the set of locked ones, and records them in the Σ^- set.

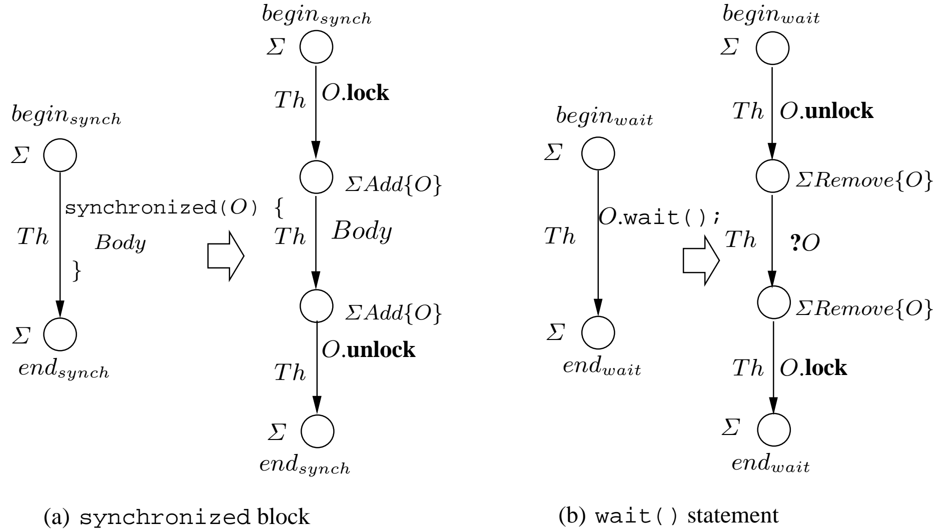

Fig. 3. Graph rewrite rules for the generation of models

Our model generator constructs a model from the source code by applying graph rewriting rules on the control flow graph. A segment P of sequential source code is modeled by a state labelled $begin_P$ denoting the control state preceding the execution of P, a transition labelled by P, and a state labelled end_P denoting the control state following P. The translation could be kept at this abstraction level or may be refined recursively. Therefore, the granularity of the model can be controlled by the designer. The translation of control-flow statements (*e.g.*, `;`, **if–then–else**, **while**, $\ldots$) is done according to standard rewriting rules. Synchronization statements are treated specially though. The `synchronized` statement (characterized by the requested object O) is translated as shown in Fig. 3(a). Entry in the `synchronized` block is modeled by a transition labeled with a **lock** on object O, while the exit is modeled by a transition labeled with an **unlock**. The fact that the thread Th holds the lock of O is recorded by adding $\{O\}$ to the Σ of the `synchronized` statement. An invocation of **wait** on O is translated by three transitions (Fig. 3(b)): one modeling the lock release and leading to a waiting state, another labelled by a *reception-of-notification* action, and a final transition modeling the lock request. These graph-rewrite rules allow us to obtain an extended automaton at the desired level of abstraction for each application thread. The information encoded in the synchronization context Σ of each state of these automata is used for informing the scheduler synthesis program about the resources which are used at the states of a thread. They are also used for constructing a *resource allocation graph* which is subsequently used for deriving a set of initial constraints against the deadlocks of the system. These constraints can be used as an initial scheduler, so as to decrease the possible behaviors of the system.

3 Scheduler Architecture and Semantics

3.1 Architecture

The architecture of the schedulers we synthesize consists of two three-layered stacks [7], as shown in Fig. 4. The left stack selects a thread for execution. The right stack selects a thread for the reception of a notification. Being able to control which thread will be notified for a particular event is something that other scheduling policies do not offer, since they concentrate only on the selection of threads for execution. After one of the scheduler stacks is finished, it passes control to an underlying *R-T OS* which provides low-level kernel mechanisms such as thread creation, suspension and resumption, and timeout enabling/disabling.

Controlling the Executing Thread The left stack takes control of the system when the application calls one of **lock**, **unlock**, **waitForPeriod**, **wait** and **waitTimed**. In these cases, it must choose one of the available threads as the thread which should be executed next. It does this in three steps, each one performed at a different layer. In the first layer, referred to as the *Ready-Exec* scheduler, it calculates the set of threads $\mathcal{R}_{\text{exec}}$ which are ready to execute without directly blocking due to mutual exclusion. The *Safe-Exec* scheduler layer is responsible for calculating the subset $\mathcal{S}_{\text{exec}}$ of $\mathcal{R}_{\text{exec}}$, consisting of those threads which can *safely* execute, that is, their execution will not cause a deadlock

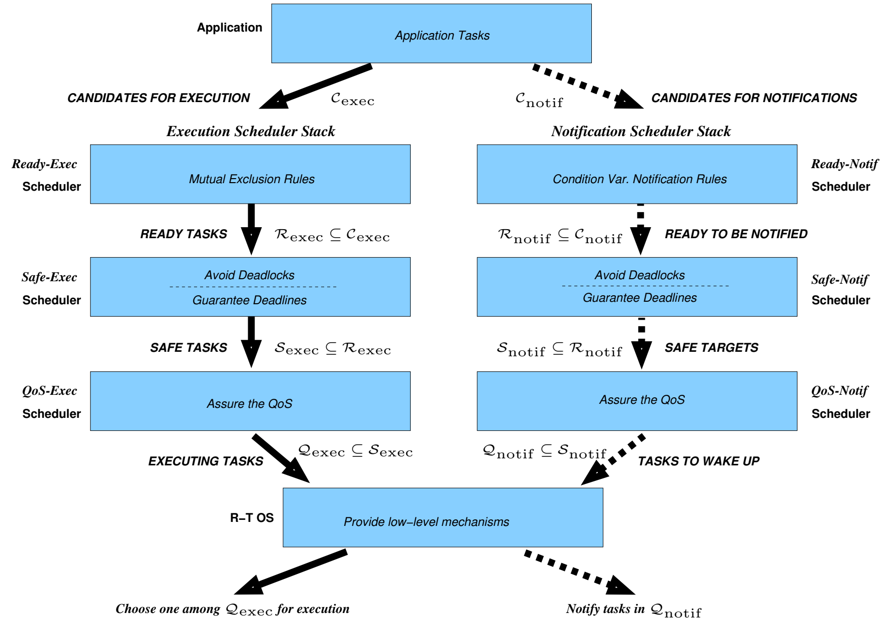

Fig. 4. Scheduler architecture

nor a deadline miss. The third layer *QoS-Exec* calculates the set $\mathcal{Q}_{\text{exec}} \subseteq \mathcal{S}_{\text{exec}}$, consisting of those *safe* threads which also respect the QoS requirements of the application.

Controlling the Notified Thread The right stack is passed control when the application calls **notify** or **notifyAll**. The reason for this is that the threads which will be notified (if any) cannot ever be selected for execution. This is because they will immediately try to re-enter the monitor after being notified and thus get blocked by the notifier (which is already in the monitor). Nevertheless, we can control which among the threads waiting for the notification should receive the notification, if there are more than one threads waiting and we are not performing a **notifyAll**. Thus, the top layer *Ready-Notif* calculates the set $\mathcal{R}_{\text{notif}}$ of threads waiting on the condition variable being notified. The middle layer *Safe-Notif*, calculates the subset $\mathcal{S}_{\text{notif}}$ of $\mathcal{R}_{\text{notif}}$ consisting of those threads which, if notified, will not cause the system to enter into a deadlock state or to miss a deadline. The *Safe-Notif* layer passes the $\mathcal{S}_{\text{notif}}$ set to the bottom *QoS-Notif* layer, which calculates the subset $\mathcal{Q}_{\text{notif}}$ of $\mathcal{S}_{\text{notif}}$, consisting of the threads which can be safely notified and also respect the QoS requirements.

QoS Policies The complexity of the QoS layer is controlled by the application designer. In choosing a QoS policy (or policies, since these are composable) the designer can balance between the execution time and extra memory space needed by the policy and the gains to the overall system quality the particular policy can offer. A QoS policy is,

for example, the *minimization of the response jitter of some thread* (*e.g.*, if it controls a physical device), or the *local minimization of context switches* (LMCS) in order to speed-up the execution and (hopefully) minimize cache misses/flushes and, thus, also energy consumption. This latter policy can be implemented quite easily, since all one needs to examine is whether the currently executing thread T_{Exec} is in the set $\mathcal{S}_{\mathrm{exec}}$ of threads which are safe to execute next. If this is the case, then we can let it continue its execution, by setting the set $\mathcal{Q}_{\mathrm{exec}}$ equal to the singleton $\{T_{\mathrm{Exec}}\}$.

Other, more complex policies may take their decisions by examining application variables and/or (parts of) the execution history.

3.2 Semantics

The model of the system we construct is the parallel composition of: *(i)* an automaton which is responsible for advancing time and firing timeouts, *(ii)* one automaton for each of the application threads, and *(iii)* two more automata, for the *QoS-Exec* and the *QoS-Notif* scheduler layers respectively. The application automata are those obtained from the Java source code, appropriately annotated with the timing constraints modeling the execution times of the code that has been abstracted away.

The state of the system model comprises of:

- a program counter (PC_i) for each of the application threads,
- a local clock (C_i) for each thread, which is used for their computations and the timeouts if they execute a **waitTimed**,
- a global clock (C_{System_i}) for modeling the periods of each periodic thread,
- a variable (T_{Exec}) holding the currently executing thread,
- two boolean variables (*Exec_Sched_Enabled* & *Notif_Sched_Enabled*) for controlling whether it is one of the scheduler automata (and which one of them), or the time (when they are both **true**) or the time and application automata (when they are both **false**) which should execute, and
- the *boolean* variables of the application threads used in conditionals associated with waiting statements.

The system goes through three different modes of execution, as shown in Fig. 5(c). In the "Time Only" mode (where *Exec_Sched_Enabled* $=$ *Notif_Sched_Enabled* $=$ **true**) the automaton responsible for advancing time and firing timeouts (shown in Fig. 5(a)) is the sole automaton enabled in the system and it can fire one or more timeouts, if any is enabled, corresponding to the expiration of a **waitTimed** or **waitForPeriod**. If a timeout is fired then the execution mode changes to "Schedulers Only" (where *Exec_Sched_Enabled* $= \neg$*Notif_Sched_Enabled*), so that our scheduler can handle it. If there is no timeout to be fired then the execution mode changes to "Time and Application" (where *Exec_Sched_Enabled* $=$ *Notif_Sched_Enabled* $=$ **false**). At this mode, both the time automaton and the automata of the application are enabled. If the application automaton needs to execute a time guarded action (*i.e.*, a computation), then it blocks, allowing time to advance. The time automaton then performs a tick (*i.e.*, a time step) and we pass back to the "Time Only" mode, so as to check if a timeout has now been enabled. If, however, an application automaton needs to perform an action which causes re-scheduling, then it passes control back to the schedulers (*i.e.*, the mode now becomes "Schedulers Only").

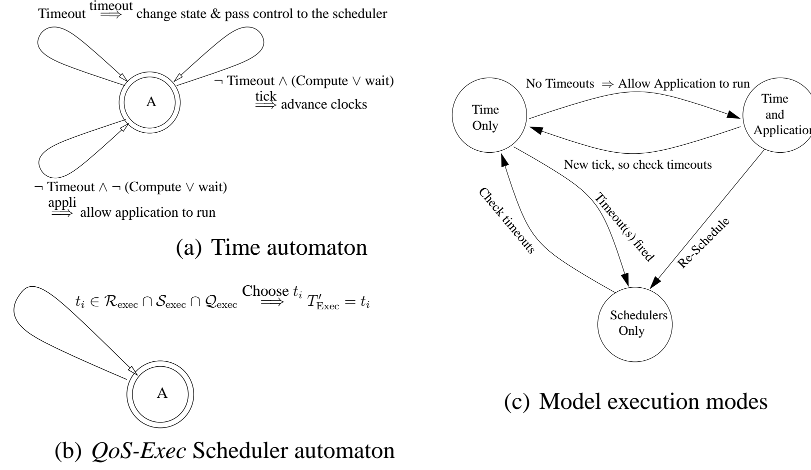

Fig. 5. Time & Scheduler automata and system execution modes

4 Scheduler Synthesis

In order to synthesize the *Safe-Exec* and *Safe-Notif* scheduler layers, we first construct the set of reachable states and, thus, identify the deadlocks. These are the states where the application threads are deadlocked, or the states where some thread has missed its deadline or period (since in that case we block the system explicitly). The existence of these states indicates that the predicate we are currently using to describe the set $\mathcal{S}_{\text{exec}}$ (resp. $\mathcal{S}_{\text{notif}}$) needs to be constrained even further. This predicate starts with the value of **true**, thus accepting initially all threads in the set $\mathcal{R}_{\text{exec}}$ (resp. $\mathcal{R}_{\text{notif}}$) as safe, where $\mathcal{R}_{\text{exec}}$ has been calculated during the extraction of the model from the source code. Having obtained the deadlocked states, we do a backwards traversal of the whole state space starting from the deadlocked states, until we reach a state which corresponds to a choice of one of the scheduler automata. There, we identify the choice $T_{\text{Exec}} = t_i$ which allowed the path leading to a deadlock state(s) and create a new constraint for the layer *Safe-Exec* (resp. *Safe-Notif*). This constraint is constructed by changing the set $\mathcal{S}_{\text{exec}}$ (resp. $\mathcal{S}_{\text{notif}}$) to be:

$$\mathcal{S}'_{\text{exec}}(\overrightarrow{state}) = \mathcal{S}_{\text{exec}}(\overrightarrow{state}) \setminus \{t_i\}$$

If at some point we find that $\mathcal{S}'_{\text{exec}}(\overrightarrow{state})$ is equal to the empty set, then we add the current state to the set of deadlocks and continue the synthesis procedure.

4.1 State-Space Reduction and Application Analysis

Even though the basic idea of synthesizing the *Safe-Exec* and *Safe-Notif* scheduler layers is simple, it is evident that in practice it suffers from the state explosion problem. Therefore, it is imperative that we use techniques to minimize the size of the state space. Our method consists of synthesizing schedulers for successively more detailed models,

adding thus complexity to a model only when we have already calculated how we can constrain the more abstract one. The scheduler synthesis is performed in five major steps.

Compositional Synthesis First, we decompose the system and synthesize constraints independently for each of the components. We then apply the synthesis algorithm again on the parallel composition of the already constrained models. In the case study, we decided to decompose the application in two sub-systems, one consisting of 4 threads, namely, `Lifter`, `Putter`, `SensorReader`, and `TrajectoryControl`, and another one comprising the `Controler` thread. The decision was mainly taken because of the size of the corresponding models (see Table 1). This approach allows us to start the synthesis with a model about 84% smaller than the original one.

Table 1. Model abstractions and optimizations

In the "original" models the `IDLE` task is allowed to execute only when no other task is safe. Otherwise, the state-spaces explode - the 5-thread system has more than 404 M states.

Model kind	States	Red.	Trans.	Red.	Dead.
Model Abstractions & Optimizations for 4 threads					
T ***original* (i.e., *P*)**	92382	0.00%	103658	0.00%	61
U	3320	96.41%	4680	95.49%	0
T ***NP***	74650	19.19%	83018	19.91%	61
T ***P, bbe***	20866	77.41%	25020	75.86%	1
T ***NP, bbe***	18304	80.19%	21738	79.03%	1
Model Abstractions & Optimizations for 5 threads					
T ***original* (i.e., *P*)**	600086	0.00%	695653	0.00%	1814
U	5080	99.15%	8232	98.82%	0
T ***NP***	445979	25.68%	511809	26.43%	1579
T ***P, bbe***	221750	63.05%	271429	60.98%	1
T ***NP, bbe***	171238	71.46%	206655	70.29%	1

Abstraction of Time Second, we consider the issue of *time*. We examine the *untimed* model (U) of the system and search for constraints which can guarantee the *absence of deadlocks*. Searching for deadlocks in the untimed model allows us to examine a much smaller search space. In the case study we observed a reduction of 96% for the 4-thread subsystem and 99% for the whole system (see Table 1). More importantly, finding and removing *all* deadlocks in the untimed model means that the application is *logically* correct. An initial set of deadlocks can actually be obtained during the generation of the model from the source code by applying standard analysis techniques for deadlock detection (typically a search for loops in a dependency graph). Our model generator implements such an analysis as well.

Having found all the potential deadlocks in the untimed system, we add the synthesized $\mathcal{S}_{\text{exec}}$ and $\mathcal{S}_{\text{notif}}$ scheduler sets obtained so far to the *timed* model (T), in order to

search for the *timeliness* constraints, which can guarantee that all threads will meet their deadlines. In order to make the problem more tractable, we reduced the timed model modulo the *branching bisimulation equivalence (bbe) reduction* [16], which eliminates "unobservable" actions (in our case the Tick action) but only when doing so preserves the branching structure of processes. Given a set of equivalent states under the *bbe* reduction, we elect as a representative of this set the state which has the maximum global clock value.

Execution Model Third, we analyze the behavior of the system for two different execution models, namely *preemptive* and *non-preemptive*. We first consider that the application threads cannot be preempted while they are computing. The non-preemptive execution model hypothesis reduces the state space, since it removes all the cases where the execution of a thread is suspended so as to handle an interrupt. Once we can indeed safely schedule the system under the hypothesis that threads are never preempted, then we can use the constraints obtained during this step to *reduce even further* the state space that we have to construct and analyze when we do allow threads to be preempted. The non-preemption of threads is easily added to our models *through the use of a QoS policy* that forbids the schedulers from choosing a thread for execution, when another thread is already in a state where it is performing a computation:

$$\mathcal{Q}_{\text{exec}}(\overrightarrow{state}) = \{t \,.\, t \in \mathcal{S}_{\text{exec}}(\overrightarrow{state}) \wedge \neg\exists t' \neq t \,.\, \text{computes}(t')\}$$

In the case study, the combination of the non-preemptive execution model (NP) with the *bbe* leads to a reduction of about 80% for the 4-thread subsystem and 70% for the 5 threads. However, we cannot safely schedule all systems when we do not allow threads to be preempted. This means that for these systems we will not obtain any scheduling constraints and, therefore, will be obliged to examine the larger, unconstrained state space of the timed model, which corresponds to a preemptive execution model.

Observability of Clocks Having synthesized a safe scheduler for an application does not necessarily mean that we can implement it easily with an existing *R-T OS* though. The difficulty of implementing it as is, arises from the fact that the constraints we produce during the synthesis use the state of the system to decide what are the safe choices at each point during the execution and, therefore, also make reference to the values of the local clocks of the threads. However, these clocks do not really exist in the application but were only introduced as a way to model the computations of the threads. Introducing them in the final code means that we will have to add for each thread an additional timer object and reset and activate (resp. deactivate) the timer before (resp. after) each computation and read its value when making a scheduling decision. As using timers may substantially increase the execution time of the scheduler, we investigate the possibility of synthesizing a clock-free one, which only examines the *PC*s of the threads. This will make the scheduler itself faster to execute, since in order to make a scheduling decision it now only needs to examine the n values of the different *PC*s and not the $2n + 1$ values of the *PC*s, the local thread clocks and the global clock.

On the other hand, removing the clocks from the constraints can introduce states where the scheduler will take the wrong decision and cause a thread to miss its deadline.

These states are those where a scheduler gets called at the same configuration of thread *PC*s but at different time instances. Since the time instances (and therefore the clock values) are different, the safe sets $\mathcal{S}_{\text{exec}}$ of these states can be different themselves as well. When we decide to not observe the clock values while scheduling, we are effectively unable to differentiate among these different sets and all these states become *equivalent*, as far as our scheduler is concerned. Therefore, if we wish our scheduler to always make a decision which is *safe*, then the $\mathcal{S}_{\text{exec}}$ set of this *equivalence class* of states should be the *intersection* of the $\mathcal{S}_{\text{exec}}$ sets of the states which belong to the same equivalence class.

$$\mathcal{S}_{\text{exec}}(\overrightarrow{class_j}) = \bigcap_{state_i \in class_j} \mathcal{S}_{\text{exec}}(\overrightarrow{state_i})$$

Sometimes, the $\mathcal{S}_{\text{exec}}(\overrightarrow{class_j})$ set will be empty, if the scheduler decisions at the members of this equivalence class were conflicting. When encountering such an equivalence class whose $\mathcal{S}_{\text{exec}}$ set is the empty set, we need to add its members to the set of deadlocked states and continue the synthesis algorithm, until we find a set of constraints which helps us to avoid the whole class, if any.

Other QoS Policies Once we have synthesized a safe scheduler, we can compose it with other QoS policies to choose among the safe threads those which better realize the QoS requirements of the system. Actually, as Altisen *et al.* [2] showed, a QoS policy can also be used from the start of the synthesis as an initial *scheduling policy*, so as to reduce the size of the model we want to analyze. In this case, the QoS policy used should give preference to tasks of the system which have a higher probability to miss their deadlines.

Table 2. Synthesis steps

Model kind	**States**	**Red.**	**Trans.**	**Red.**	**Dead.**	**Constr.**
Synthesis Steps for 4 threads						
U	3320	96.41%	4680	95.49%	0	0
U, No Deadlocks	3320	96.41%	4680	95.49%	0	0
T ***NP, bbe*** , No Deadlocks	18304	80.19%	21738	79.03%	1	0
T ***NP, bbe*** , Safe	13830	85.03%	16196	84.38%	0	15
T ***P, bbe*** , No Clocks	16807	81.81%	20003	80.70%	1	15
T ***P, bbe*** , No Clocks, Safe	16249	82.41%	19245	81.43%	0	23
Synthesis Steps for 5 threads						
T ***LMCS, bbe*** , 4 threads Safe	23302	96.12%	26515	96.19%	1	23
T ***LMCS, bbe***	15742	97.38%	17584	97.47%	0	38
T ***LMCS, bbe*** , No Clocks (Safe)	15742	97.38%	17584	97.47%	0	38

Table 2 shows the results obtained when applying our methodology on the case study. First we synthesized a safe scheduler consisting of 15 constraints for the 4-thread

subsystem, considering that the execution mode is non-preemptive. Then we used these constraints to render the state space smaller once we choose a preemptive execution mode and we no longer observe clocks. This lead to another 8 constraints, with which the 4-thread subsystem is indeed always safe. Then, we used the 23 constraints we synthesized for the 4-thread subsystem, to minimize the 5-thread one. In parallel, we also used a QoS policy which locally minimizes context switches (LMCS), so as to render the state space even smaller. This produced another 15 constraints, under which the 5-thread system is safe, when it is running with the LMCS QoS policy.

5 Scheduler Implementation

Once the scheduler has been synthesized using the model, it has to be implemented and integrated with the code generated by the TurboJ compiler. The structure of the executable code is depicted in Fig. 6. The generated code consists of two parts, namely, the application code and the synthesized scheduling predicates. The application code is instrumented to call the application-level scheduler called `J_Scheduler` when an application thread executes the code corresponding to the Java-bytecode statements **monitorEnter** or **monitorExit**, and the methods **notify**, **notifyAll**, **wait**, **waitTimed** and **waitForPeriod**. `J_Scheduler` calls the function `Synthesized_Predicates` which evaluates the synthesized predicates corresponding to the different scheduler layers. The application-level scheduler is implemented on top of an accompanying runtime library, which offers an implementation in native code of the aforementioned statements and methods.

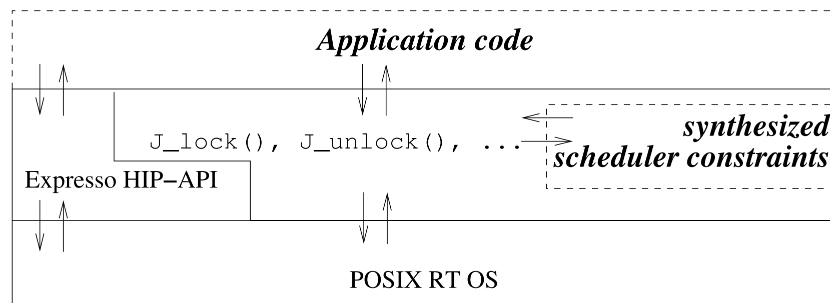

Fig. 6. System decomposition

More precisely, our implementation uses a subset of POSIX, *i.e.*, it uses mutexes (without any priority inheritance protocol), condition variables & priorities. Indeed, it does not use real-time timers, since one of the underlying OS's we needed to support (in the context of the Expresso project) does not provide any. More specifically, we use a single mutex (`sched_mx`) and provide to each application thread a unique condition variable. These condition variables are all associated with the aforementioned mutex (a capability which exists in POSIX but not in Java). Finally, we use three different

Table 3. Pseudo-code of the application scheduler

```
1  void J_Scheduler(int tc, bool in_notify, timespec &deadline) {
2    bool finished = true, level_super = false;
3    int  tn;
4    /* tc is the current thread (t_c)& tn the next one (t_n)*/
5    do {
6      tn = Synthesized_Predicates(THREADS_TABLE); // calculate Ready, Safe & QoS sets
7      if (tn != tc) {
8        if (in_notify) {
9          if (-1 != tn) { // -1 means no thread is waiting
10           J_Set_Priority(tn, BLOCKED); // Don't allow t_n to preempt you
11           notify(THREADS_TABLE[tn].cv);
12         }
13       } else { // ! in_notify
14         J_Set_Priority(tn, EXEC);
15         THREADS_TABLE[tn].position = THREADS_TABLE[tn].pos_after_notif;
16         notify(THREADS_TABLE[tn].cv);
17
18         if (NULL == deadline) { // Not a waitTimed or waitForPeriod
19           // Release CPU here
20           wait(THREADS_TABLE[tc].cv, sched_mx);
21           /* Here I have been signaled  */
22         } else {             // NULL != deadline
23           J_Set_Priority(tc, SUPER); // Release CPU here
24           timed_wait(THREADS_TABLE[tc].cv, sched_mx, deadline);
25           /* Here I have been signaled or I have timed-out.
26              Must re-schedule to be safe, if I timed-out. */
27           if (THREADS_TABLE[tc].position != THREADS_TABLE[tc].pos_after_notif) {
28             finished = false;
29             THREADS_TABLE[tc].position = THREADS_TABLE[tc].pos_after_timeout;
30           }
31           level_super = true ; deadline = NULL;
32         }
33       }
34     }
35   } while (! finished);
36   if (level_super) J_Set_Priority(tc, EXEC);
37 }
38
39 void J_lock(int tc, int curr_pos) { // J_unlock is exactly the same
40   lock(sched_mx);
41   THREADS_TABLE[tc].position = curr_pos;
42   J_Scheduler(tc, false, NULL);
43   unlock(sched_mx);
44 }
```

priority levels, namely, **BLOCKED**, **EXEC** & **SUPER** (from lowest to highest) and the SCHED_FIFO POSIX scheduling policy (*i.e.*, priority based, FIFO, non-preemptive execution of tasks having the same priority).

The pseudo-code of this implementation is shown in Table 3. Before calling `J_Scheduler`, the running application thread locks `sched_mx` and changes the label used for marking its position in the model. `J_Scheduler` calls the function `Synthesized_Predicates` (generated by the synthesis tool) passing it the current labels of all the application threads. If the thread chosen to be executed next (t_n) is different from the current one (t_c) and t_c is not doing a notification, we set the priority of t_n to **EXEC**, we notify the condition variable of t_n (cv_{t_n}) and finish by having t_c wait on its own condition variable (cv_{t_c}). This final action releases `sched_mx` just before blocking, thus allowing the notified thread t_n to resume execution. If t_n happens to be the same as t_c, then the application scheduler returns normally and t_c unlocks `sched_mx`.

The algorithm changes somewhat when calling the application scheduler because of a **waitTimed** or a **waitForPeriod**. In this case, we also pass to our scheduler the time that the current thread should wait. The scheduler then performs a timed wait on cv_{t_c} using as timeout the absolute deadline passed as an argument, instead of doing a simple wait. In addition, it also increases the priority of t_c to **SUPER** just before performing the timed wait, so that t_c has the chance to get the CPU as soon as it timeouts. If t_c does indeed timeout, it re-calculates the scheduler predicates so as to find out if it is indeed safe to continue execution. Functions `J_timed_wait` and `J_wait` set the field `THREADS_TABLE[].pos_after_notif` right before calling function `J_Scheduler` to be the label corresponding to the internal state of the wait where the thread has been notified but has not yet reacquired the mutex of the object on which it was waiting. In a similar manner, the field `pos_after_timeout` is set by the functions `J_timed_wait`, and `J_wait_for_period` to the label corresponding to the internal state of the **waitTimed**, or the label corresponding to the first statement following the beginning of a new period.

We have successfully executed our implementation over two different combinations of hardware architecture and embedded OS's, namely an Intel Pentium II (333MHz) running `eCos` over Linux and a PowerPC simulator (PSIM) using the proprietary OS of an industrial partner of the Expresso project. The experiments we performed with `eCos` showed that the execution time of the synthesized predicates (*i.e.*, the execution time of function `Synthesized_Predicates` called by `J_Scheduler`) is comparable to the execution time of locking an (unlocked) mutex, having a WCET of the order of 4μs.

Besides, we have a non-POSIX prototype implementation over `eCos` that uses alarms and alarm handlers, thus allowing us to support deadline and period miss handlers as proposed by the RTSJ, but disallowed by the Expresso and the Ravenscar-Java profiles.

6 Conclusions

We have presented a complete application-driven scheduler synthesis chain that allows to automatically generate native code for embedded real-time systems based on Java. In addition, it allows one to substitute RMA/EDF & PCP with a scheduler which is still safe but not as pessimistic and which can be easily extended with QoS scheduling policies. We are not aware of any other similar chain. The full integration of the model-

generation and scheduler-synthesis tools with the compiler is under test as it requires a close collaboration with the TurboJ development team.

References

1. AICAS. http://www.aicas.com/jamaica.html. JamaicaVM.
2. K. Altisen, G. Göessler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Real-Time Systems*, 23(1), 55–84, July 2002.
3. B. Delsart, V. Joloboff, and E. Paire. JCOD: A Leightweight Modular Compilation Technology for Embedded Java. In *EMSOFT'02*, Grenoble, France, October , 2002.
4. L. Gauthier and M. Richard-Foy. Expresso RNTL Project - High Integrity Profile, 2002. Available from http://www.irisa.fr/rntl-expresso/docs/hip-api.pdf
5. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, USA, 1996.
6. IEEE. POSIX.1. IEEE Std 1003.1:2001. Standard for Information Technology - Portable Operating System Interface (POSIX). The Institute of Electrical and Electronic Engineers, 2001.
7. Ch. Kloukinas and S. Yovine. Synthesis of Safe, QoS Extendible, Application Specific Schedulers for Heterogeneous Real-Time Systems. In *Proceedings of 5th Euromicro Conference on Real-Time Systems (ECRTS'03)*, Porto, Portugal, July 2003.
8. J. Kwon, A. J. Wellings, and S. King. Ravenscar-Java: A High-Integrity Profile for Real-Time Java. In *Java Grande*, pp. 131–140, 2002.
9. M. Lusini and E. Vicario. Static analysis and dynamic steering of time-dependent systems using Petri Nets. Technical Report # 28.98, University of Florence, 1998.
10. G. Muller, B. Moura, F. Bellard, and Ch. Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proc. of Usenix COOTS'97*, Berkeley, 1997.
11. Real-Time for Java Expert Group. The Real-Time Specification for Java, 2001. Available from http://www.rtj.org
12. Real-Time Java Working Group. Real-Time Core Extensions, revision 1.0.14, 2001. Available from http://www.j-consortium.org/rtjwg.
13. J. Sifakis, S. Tripakis, and S. Yovine. Building models of real-time systems from application software. In *Proceedings of the IEEE, Special issue on modeling and design of embedded software*, 91(1):100-111, January 2003. IEEE.
14. Sun Microsystems. The Java HotSpot Performance Engine Architecture. http://java.sun.com/products/hotspot/whitepaper.html, April 1999.
15. TimeSys. http://www.timesys.com. JTime.
16. R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *JACM*, 43(3), 1996.
17. M. Weiss, F. de Ferrière, B. Delsart, Ch. Fabre, F. Hirsch, E. A. Johnson, V. Joloboff, F. Roy, F. Siebert, and X. Spengler. TurboJ, a Java bytecode-to-native compiler. In *Proc. of LCTES'98*, volume 1474 of *LNCS*, 1998.