RTComposer: A Framework for Real-Time Components with Scheduling Interfaces

Rajeev Alur and Gera Weiss
University of Pennsylvania
alur@cis.upenn.edu, gera@seas.upenn.edu

ABSTRACT

We present a framework for component-based design and scheduling of real-time embedded software. Each component has a clearly specified interface that includes the methods used for sensing, computation, and actuation, along with a requirement given as a regular set of macro-schedules. Each macro-schedule is an infinite sequence that specifies, for every time slot, the set of component methods invoked in that slot. The macro-scheduler composes the specifications of all the components, along with the platform specification that constrains which methods can be executed within a single slot, to generate a feasible macro-schedule. Within a slot, we use logical execution time semantics, and this micro-scheduling is implemented on top of a native priority-based scheduler. With this approach, each component can be specified and analyzed in a platform-independent way, and at the same time, the performance can vary with changing load and changing processing speed. We describe an implementation using Real-Time Java. Scheduling specifications can be given as periodic tasks, or using temporal logic, or as omega-automata. Components can be added dynamically, and non-real-time components are allowed. We demonstrate the benefits of the approach using case studies.

Categories and Subject Descriptors: D.2.2 [Software Engineering] : Design tools and techniques; J.7 [Computers in other systems] : Real time, Process control.

General Terms: Performance, Reliability.

Keywords: Real Time Specification for Java (RTSJ), Automata based scheduling.

1. INTRODUCTION

Components with clearly specified APIs, such as Java library classes, allow designers to build complex systems effectively in many application domains. The key to such modular development is that an individual component can be designed, analyzed, and tested without the knowledge of other components or the underlying computing platform. When the system contains components with real-time requirements, the notion of an interface must include the requirements regarding resources, and existing programming languages provide little support for this. Consequently, current development of real-time embedded software requires significant low-level manual effort for debugging and component assembly (cf. [15, 19, 26]). This has motivated many researchers to develop compositional approaches and interface notions for real-time scheduling (cf. [4, 8, 9, 23–25, 27, 29]).

The most common way of describing the usage requirements of a real-time component is to specify a period, sometimes along with a deadline, which gives the frequency at which the component must execute. The designer of the component makes sure that the performance objectives will be met as long as the component is executed consistent with its period. For implementation, the real-time operating system performs a worst-case execution time analysis on all the components, followed by schedulability analysis to check whether all the timing requirements can be met (cf. [6, 17, 20]). Specifying resource requirements using periods has advantages due to simplicity and analyzability, but has limited expressiveness. For example, a specification such as "execute the component every 5ms" does not say whether the scheduler should or should not execute it more frequently if enough computing resources are available, and if a component has multiple methods, say, for different control tasks, each needing a different period, the requirement cannot be naturally captured by a single period.

In this paper, we use finite automata over infinite words as an expressive, analyzable, and composable specification framework for resource requirements [2, 32]. specification framework for resource requirements [2,32]. We assume that the resource is allocated in discrete slots of some fixed duration in the style of time-triggered architecture [18] and the LET (Logical Execution Time) programming abstraction [13]. The interface of a real-time component consists of (1) a logical slot length, (2) a sensing method to read inputs, (3) an actuation method to write outputs, (4) a set M of computation methods the component supports, and (5) a scheduling requirement described by a finite-state automaton A. The scheduler invokes, within each logical slot i, the method to read inputs at the beginning of the slot, a subset M_i of computation methods M, and the method to write outputs at the end of the slot. The scheduled computation, from the perspective of the component, is then a sequence of $M_1 M_2 \ldots$ of sets of computation methods invoked by the scheduler, and the requirement automaton A describes which of these sequences are acceptable to the

EMSOFT'08, October 19–24, 2008, Atlanta, Georgia, USA.

component. Each component has a well-defined semantics, and can be designed and analyzed in isolation in a platform-independent manner. However, since our scheduling specification allows multiple scenarios, the real-time performance will vary depending upon the presence of other components and platform characteristics.

Given a set of components, the scheduling decision consists of two stages. The *macro-scheduler* chooses the sequence of methods to be called within each slot. The macro-scheduler constructs the product of the automata describing requirements of all the components, and intersects it with constraints imposed by the platform that capture which methods can be co-scheduled within each logical slot. If this product automaton is empty, the components are unschedulable on this platform, and otherwise, the macro-scheduler chooses a feasible schedule. *Micro-scheduling* of the chosen methods, along with the handling of interrupts and background processing of non-real-time components, within a slot is done by the native priority-based operating system.

We describe a toolkit, RTComposer, that is implemented on top of Real Time Java [5]. The components are described as Java classes with methods corresponding to our notion of an interface. In particular, the automaton for scheduling specification can be described directly, but we also support a variety of alternative high-level specifications. These include classical periodic specifications, temporal logic formulas, and also performance objectives, such as stability, for controllers described in Simulink. The constraints imposed by the processing speed of the platform can be described by an automaton over all possible computation methods of real-time components. A more convenient way is for each component to declare the worst-case execution time for each method, in logical units, and then, specify the platform by a scaling constant. The macro-scheduler runs as a high-priority Real-Time Java thread, that wakes up at the beginning of each slot and invokes all the chosen methods before going to sleep. The implementation also allows admission of new real-time components. When a new component is added, the macro-scheduler computes the new policy by taking the product of the current policy with the scheduling specification of the new component, and checking for emptiness.

To illustrate the benefits of the proposed framework, we present a case study involving two simulated control systems along with background tasks. Each control system has two computation methods, with different computational requirements, and leading to different qualities for estimation of plant state. The scheduling specification says that one of the two methods must be executed in every logical slot, and the more precise, and computationally more demanding, estimation must be executed at least once every two logical slots. With this case study, we demonstrate how RTComposer allows dynamic schedules. Specifically, we study three types of dynamic adaptations. First, we experiment with dynamic load conditions and show that our approach allows safe degradation of control performance when resources are needed for other computations. Second, we demonstrate how our approach allows platform portability in a way that allows better performance on faster machines. The third type of dynamic adaptation that we study is online admission of components. We show that, when a component is not installed, other components can use the resources that it does not use to improve their performance.

2. AN ILLUSTRATIVE EXAMPLE OF MULTIPLE CONTROLLERS

As an illustrative example, consider two independent plants controlled by controllers that share a single CPU, as depicted in Figure 1. In addition to the controllers that use the CPU to compute a feedback to the plants, the CPU is also used by non real-time background applications and sporadic interrupt handlers. This is a typical situation in many modern application where real-time computations are combined with other functionalities.

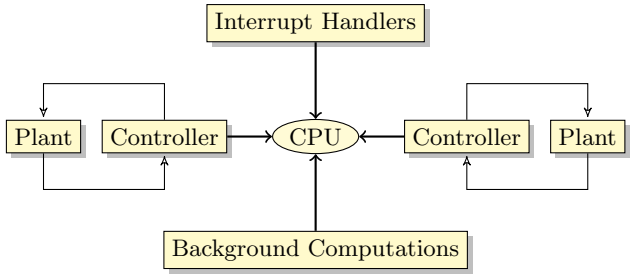

Figure 1: Two real-time applications sharing a CPU with other tasks.

As an example, we show how RTComposer can be used to implement the controllers in a way that allows platform independent performance guarantee, dynamic adaptation to variable resources availability and online admission.

Towards a resource-aware implementation, two modes of operation are designed for each controller, as follows. In both modes the controller updates its internal state, which is an estimation of the state of the plant, and then uses this estimation to compute a feedback to regulate the plant. In the first mode the estimation is updated using the output of the plant (by, e.g., the Linear Quadratic Regulator (LQR) control design). In the second mode, a lightweight simulation of the plant is used. This design is motivated by applications where either obtaining the output of the plant or processing the estimation procedure are computationally demanding.

```
class Controller implements Component {
  ...

  // Heavy computation
  @WCET(0.8)
  public void estimate() {
    ...
  }

  // Lightweight version
  @WCET(0.1)
  public void simulate() {
    ...
  }
}
```

Listing 1: A controller with two modes.

Listing 1 shows a skeleton of an implementation of the class `Controller` (a common base for the classes `Controller1`

and Controller2). The controller is implemented as a Java class with two methods called estimate and simulate that are implementations of the two modes described in the previous paragraph.

Assume that the controllers are designed for sampling the plants every 10ms. Then, from the perspective of each controller, execution can be viewed as a sequence of time slots of length 10ms where each slot begins by reading the output and ends with setting a feedback to the plant. The state of the controller is updated in each slot, using either the estimation or the simulation modes. Such a run can be described by a sequence $f(1), f(2), \ldots$ where $f(i) \in \{\texttt{simulate}, \texttt{estimate}\}$ is the method invoked at the ith slot. This is an example the Logical Execution Time (LET) abstraction [16], discussed in more details in Section 4.2 below.

Next, we present an interface allowing real-time components, such as the two controllers, specify how methods should be assigned to slots. For example, assume that two copies of estimate cannot complete within one slot. In this case, we need to schedule Controller1.estimate at some slots and Controller2.estimate in other slots. To allow background computations, we may also wish to schedule neither at some slots.

We call the assignment of task sets to slot *macro-schedule*. Figure 2 depicts an example of a macro-schedule. In this example, at each slot, the state of one of the controllers is updated by triggering the heavy estimation while the state of the other controller is updated by a lightweight simulation.

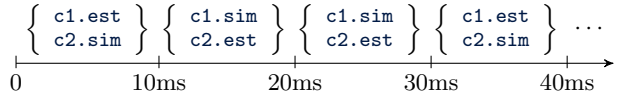

Figure 2: A macro-schedule. c1, c2, est and sim are shortcuts for Controller1 and Controller2, estimate, and simulate, respectively.

In control software, we wish to choose macro-schedules such that controllers maintain stability requirements. With RTComposer, the designer of the class Controller can provide an automaton that specifies the sequences of task sets that allow the controller to maintain stability.

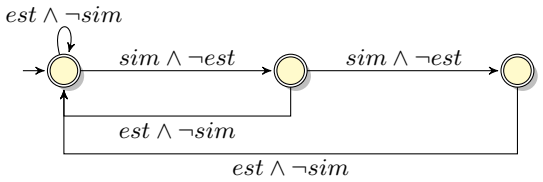

Figure 3: Automaton for class Controller. The symbols *est* and *sim* are shortcuts for estimate and simulate, respectively.

For example, consider the automaton depicted in Figure 3. When given as a scheduling specification, the automaton says that the controller can deliver the required performance if, at any slot, either estimate or simulate run, and there are no three consecutive slots in which estimate is not triggered at least once. In this example, best performance is achieved when estimate is triggered every slot without any simulate run, but this schedule does not allow other components (because estimate takes too long to compute).

Having this automaton, a code for scheduling the two controllers is depicted in Listing 2.

```
public static void main(String args[]) {
  ...

  Controller cont1 = new Controller(...);
  Controller cont2 = new Controller(...);

  Scheduler sch = new Scheduler();

  sch.addComponent(cont1);
  sch.addComponent(cont2);

  if( sch.isSchedulable() ) {
    sch.start();
  } else {
    out.println("Not schedulable");
  }
}
```

Listing 2: Scheduling two controllers.

In this code, the two controllers are instantiated and then added as components. Then, the scheduler automatically computes the intersection automaton depicted in Figure 4. Note that this automaton is an intersection of two (identical) automata given by the controllers and the constraint that two copies of simulate cannot run together. The later constraint is automatically deduced by the scheduler based on the WCET (worst case execution time) annotations provided in Listing 1 (WCET of two copies of simulate sum to more than one).

The method isSchedulable returns false iff the language of the automaton is empty. In this case, the language is not empty and the program proceeds to call sch.start(). This method selects a word in the ω-language specified by the automaton and executes the components accordingly. For example, we can select an arbitrary infinite word by simulating a random walk over the automaton. As the ω-language of the automaton is in the intersection of all scheduling specifications (which are all safety languages), we are guaranteed that the generated schedule allows good performance of all components. Since we also intersected with the platform specification, we are also guaranteed that all task sets finish before the end of their slots.

3. INTERFACE SPECIFICATION

The interface for software components is given in Listing 3. Specifically, a component is a Java class implementing the methods:

- getSlotLength: specifies the length of the logical execution slot for the component.
- getAutomaton: gives an automaton specifying a safety language over subsets of the set of methods.
- readInputs: reads data from external interface to local variables (assumed to be fast).
- writeOutputs: writes data from local variables to external interface (assumed to be fast).

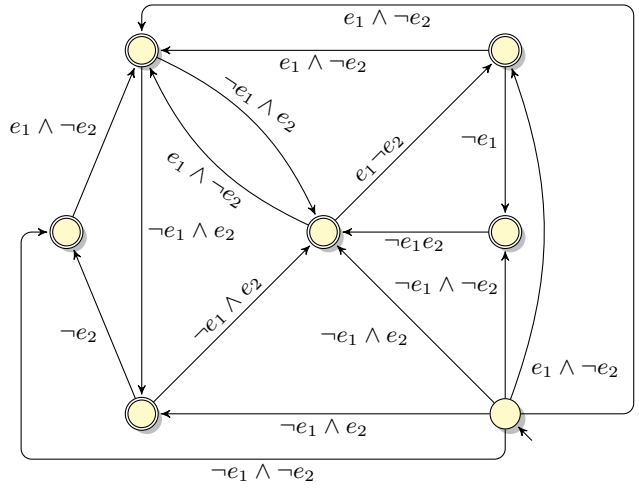

Figure 4: The product automaton. For short, we use e_1 and e_2 instead of `cont1.estimate` and `cont2.estimate`, respectively, and assume that `estimate` $\iff \neg$`simulate` is known.

```
public interface Component {
  float getSlotLength();

  File getAutomaton();

  void readInputs();

  void writeOutputs();

  // Implementation (scheduled methods)
  ...
}
```

Listing 3: The Component interface.

In addition to these four methods, the component should have a set of methods that implement its functionality (whose invocation sequence is specified by the automaton). The signature of these methods must be: `public void<name>()`. Namely, they must be public, expect no parameters, and return no value. Data can be passed from one method to another and between successive invocation of the same method via class variables.

As shown in Listing 1, an annotation called `@WCET` is implemented to allow the specification of Worst Case Execution Time for each method (if WCET is not specified, it is assumed that the execution of the method takes negligible time). This annotation specifies execution time on a reference platform. See Section 4.1, below, for a discussion of how platform portability is supported.

For specifying automata, we use the file format generated by the GOAL (Graphical Tool for Omega-Automata and Logics) package [31]. Specifically, the methods `getAutomaton`, described above, should return an XML file in the format exported by GOAL. This allows the use of GOAL for generating scheduling specifications. In particular, as GOAL provides an intuitive graphical interface and interoperability with other tools, a variety of input methods are enabled.

The semantics of the above specification are as follows. Say that `getSlotLength` returns some $\Delta > 0$. This means that, for each $i = 0, 1, \ldots$, the component expects the scheduler to invoke `readInputs` at time $i\Delta$ and `writeOutputs` at time $(i+1)\Delta$ (assuming that the execution of these methods takes negligible time). Let $S(i)$ be a subset of the methods of the component that are invoked (by the scheduler) between time $i\Delta$ and time $(i+1)\Delta$. The automaton given by `getAutomaton` specifies the set of sequences $S(1), S(2), \ldots$ that the component allows (sequences that allow the component to deliver its performance requirements).

Note that the current implementation of RTComposer assumes that slot lengths are uniform for all components. Specifically, any component whose slot length is different from the one of the first component is rejected. However, because this constraint is not inherent to the methodology, the specification interface allows different slot length for components (anticipating future versions).

4. COMPOSITION OF COMPONENTS

Assuming that a set of components is provided, each adhering to the interface described in the previous section, we discuss how to compose and run them in a way that respects the semantics of the interface.

4.1 Development process

We propose the development process depicted in Figure 5. First, the functionality and a specification of resource requirements are developed for each component. As described in the previous section, the functionality is given as a Java class and resource requirements are specified by an automaton whose alphabet is sets of methods of the class.

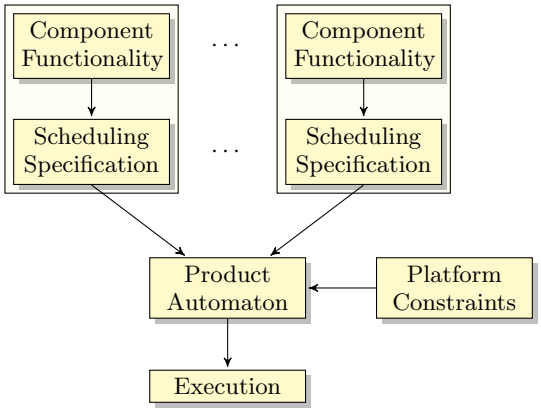

Figure 5: Development process.

To this automata, we add the platform constraints. This is also an automaton whose alphabet is the power-set of the set of methods of all components. The platform constraints automaton specifies which sets of methods can be executed in each slot. For example, if a set $M = \{m_1, \ldots, m_l\}$ of methods (not necessarily of the same component) cannot run together in any single logical execution slot, the platform constraint may include the LTL predicate `[]~(`m_1 `/\ ⋯ /\`m_l`)` (always not m_1 and m_2 and $\ldots$ m_l).

In addition to the general interface, where platform constraints are directly specified as an automaton, we propose a simpler interface, as follows. In its simplest form, when

the system runs on the reference platform, the interface is just the `@WCET` annotations. If the platform is not explicitly assigned, the scheduler assumes that the reference platform is used (the one for which the worst case execution times are computed). In this case, the scheduler would compute an automaton of the form `[] ~overload` where `overload` is a propositional formula characterizing the sets of functions whose total execution time is more than the slot length.

For platform portability, an interface for specifying how the current platform is different from the reference is also proposed. This specification comes in the form of a mapping of methods to affine functions. Given a method, the affine function mapped to it tells the scheduler how worst case execution time on the reference platform is mapped to the actual worst case execution time. For example, one can specify that all methods run twice as fast as they would run on the reference implementation. Affine functions give opportunity to add a context switch cost and a speed rate.

Having the platform specification, an automaton is computed for the intersection of all the constraints specified by the scheduling specifications of all components together with platform constraints. This automaton can be computed using known algorithms for intersecting ω-regular languages [30]. For example, the automaton in Figure 4 was automatically generated from two copies of the automaton in Figure 3 and the platform constraints described in the preceding paragraph.

Once the scheduling constraints automaton is formed, any word in the ω-regular language specified by it allows all components to maintain required performance. For example, the automaton can be used as a starting point for optimization (see [1]) or for scheduling (see Section 6, below).

Note that our approach allows the scheduler to take advantage of faster execution platform while assuring performance for slower one. To see this feature, compare our approach with tools such as Giotto [16] or Exotasks [4] where the set of tasks executed in a logical execution slot is independent of the platform. With such tools, to allow portability, one can choose long slot lengths such that completion of all tasks is guaranteed even on the slowest relevant machine. The problem with this approach is that the power of the faster machine is wasted. Using automata based specifications, RTComposer has an explicit knowledge of what the components need and what the platform is capable of, allowing different schedules for different machines.

4.2 Execution semantics

For scheduling tasks within a slot, we adopt the Logical Execution Time (LET) abstraction [16] that allows deterministic external interface even when the underlying physical execution layer introduces bounded non-determinism.

The relation between logical and physical task execution is depicted Figure 6 (adopted from [12]). The diagram depicts an execution of one task in one logical execution slot. Logically, the task is released at the beginning of the slot and terminates at the end of it, i.e., the task processes the inputs captured at the release time and its output are only made available to the outside world at the logical termination time. The actual execution, as shown in the figure, may start and finish anywhere in the logical execution slot and the execution may be interrupted by higher priority tasks. However, the external interface remains deterministic provided only that all tasks always finish within logical execution slots.

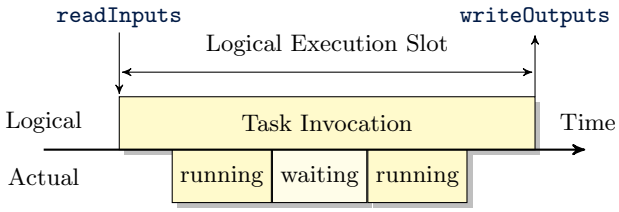

Figure 6: Execution of a task in an execution slot.

Diverging from the way LET is applied in other architectures (c.f. [4, 16]), RTComposer executes a dynamic assignment of tasks to logical execution slots. Specifically, the set of tasks that run in each logical execution slot is chosen from the intersection of all resource specifications of the components and the platform constraints specification (as described above). Particularly, improving the speed of the machine allows better performance (because less constraints means more applicable schedules and possibly existence of a better one). Similarly, we can react to variability in the background load by choosing a schedule that contains less real-time computations when the load is high. See Section 6, below, for examples.

4.3 Scheduling mechanism

To implement the above semantics, we propose a scheduling mechanism consisting of a macro (intra-slot) scheduler and a micro (inter-slot) scheduler, as follows. As depicted in Figure 7, the micro scheduler is the built-in scheduler provided by the RT-Java virtual machine. We use it to run all tasks, including interrupt handlers, real-time components and background applications. The macro scheduler mediates the execution of the real-time components, as follows. At the beginning of every logical execution slot, the macro scheduler invokes all `readInput` methods and spawns a set of tasks that wrap the methods of the real-time components to be executed in that slot. At the end of the slot, the macro scheduler invokes all `writeOutput` methods.

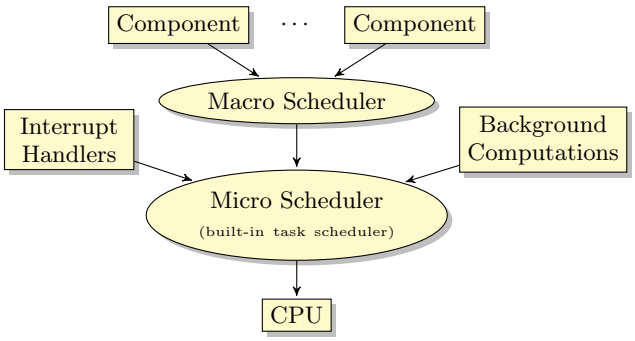

Figure 7: Micro and macro schedulers.

Priorities are assigned as follow. The lowest priority is assigned to the background applications that run only when the real-time threads and the interrupt handlers are inactive. The highest priority is given to interrupt handlers and to the macro scheduler which is considered as the highest priority interrupt handler (assuming that `readInputs` and

writeOutputs are short and deterministic). The real-time tasks, spawned by the macro scheduler, run in a priority higher than all background application and lower than interrupt handlers.

This priority assignment scheme allows coexistence of real-time tasks with both non real-time background applications and interrupt handlers, as follows. For interrupt handlers, we assume that the added latency can be bounded to allow guarantees of real-time tasks completion within allotted slots. For background application, the scheduler can spawn less real-time tasks at some times to allow more room for lower priority tasks (but not too low to disallow performance guarantee). For example, in Section 6 below, an example is given where a minimal set of real-time tasks is spawned when the background load is high and more tasks are spawned when the background load is low.

Note that our approach is independent of the algorithm used by the micro scheduler for tasks scheduling, which can be, e.g., Earliest Deadline First (EDF), Least Slack Time (LST), rate Monotonic (RM), Deadline Monotonic (DM) or any scheduling algorithm that allows guarantees of termination of all tasks before the end of each slot. More generally, the interface with the micro-scheduled does not have to be the set of tasks to run in a slot. For example, a micro-scheduler that allows task graphs to execute in slots can also fit in our framework. In this case, the alphabets of the automata are going to be task graphs (partially ordered sets of tasks) instead of unordered sets. The macro-scheduler will then compute the language of sequences of task graphs and delegate a task graph to the micro-scheduler in each slot.

4.4 Dynamic admission

A notable advantage of using automata based scheduling is that new components can be added without disturbing the continuous execution of existing ones. Specifically, when a new component is added, the existing components are assured to continue a schedule which satisfies their scheduling specifications, but possibly not the one that was planned (because the new component may not allow the one that was planned).

As a simple example, consider a component with four methods f1,f2,f3,f4 that need to execute cyclically (for all $i = 1, 2, \ldots$ and $j = 1, 2, 3, 4$, the function fj needs to execute in slot $4i+j$). Assume that this component is added and scheduled for a while. Later on, anther component is added that has only one method, f, that needs to execute in all slots. When the second component is added, the count of the running component should not be disturbed, as shown in the following figure.

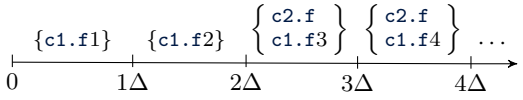

Figure 8: An addition of a component (at time 2Δ) without interrupting the running one.

In RTComposer, macro-scheduling is done by simulating a random walk over the automaton whose language is the intersection of all the scheduling specification of the components. When a new component is added, we need to start a random walk over the intersection of the running automaton and the scheduling specifications of the new component. The new random walk needs to assure that the existing components will be scheduled consistently with the current state of the running automaton.

To provide continuous schedules for the existing components, the computation of the new product automaton is done as a non real-time background computation. Also, the new product automaton should not always start executing form its initial state (as the existing components are not necessarily at their initial state). To account for that, and still provide quick turnover, a hash-table is constructed that maps the states of the running automaton to the sates of the automaton that is going to replace it. Then, when the computation of product automaton is completed, the state of the running automaton is replaced with the corresponding state of the new automaton. Removal of components can done in a similar manner.

5. SCHEDULING REQUIREMENTS

While automata based specifications are rich and intuitive, it is often useful to write the specifications in higher level languages and translate them to Büchi automata automatically. This approach simplifies specifications formulation and allows expressing requirements in a language closer to the application domain and to the relevant performance criteria. In this context, Büchi automata can be viewed as an "assembly" language to which useful specifications compile, as we show below.

```
public class Translator {
   ...

   void weakPeriodic(String predicate,
                     int    period,
                     int    offset,
                     int    deadline);

   void strongPeriodic(String predicate,
                       int    period,
                       int    offset,
                       int    deadline);

   void temporalLogic(String formula);

   void stability(Matrix[] modes,
                  String[] predicates,
                  int      h,
                  double   rho);
}
```

Listing 4: A class for translation of other specification languages to automata.

Tool support for translation of high level languages to automata is provided by the class Translator depicted in Listing 4. The methods of this class are described in the following subsections.

5.1 Periodic execution

A useful type of specifications for scheduling is the, so called, periodic task model [20]. With this model, periodic activities are defined by means of periods, deadlines and

offsets. Examples of specifying periodic constraints using the functions `weakPeriodic` and `strongPeriodic` follow.

- `strongPeriodic("p",8,1,5)`: the method `p` is invoked exactly once in slots $8i+1,\ldots,8i+6$ for every $i = 1,2,\ldots$
- `weakPeriodic("p",5,0,5)`: the method `p` must run at least once in slots $5i,\ldots,5i+4$ for every $i = 1,2,\ldots$
- `weakPeriodic("p \/ q",1,0,6)` : either `p` or `q` are invoked at least once in slots $i,\ldots,i+6$ for every $i = 1,2,\ldots$
- `strongPeriodic("p /\ ()q",7,0,0)` : `p` and `q` are invoked at times $7,14,\ldots$ and $8,15,\ldots$, respectively.

Following are some examples that show how such requirements are translated to automata. An automaton for the language `strongPeriodic("p",8,1,5)` is depicted in Figure 9 (note the similarity to Tree Communication Schedules [3]).

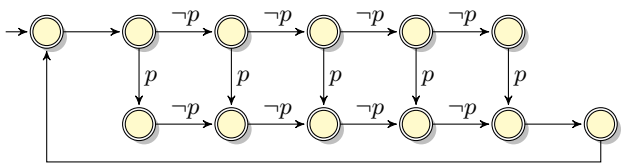

Figure 9: Strong periodic specification.

An automaton for `weakPeriodic("p",1,0,6)` is depicted in Figure 10, as an example of weak periodic specification with deadline longer than period.

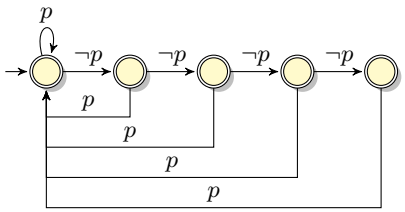

Figure 10: Weak periodic specification.

Note that `strongPeriodic("p /\ ()q",7,0,0)` could also be written as the conjunction of `strongPeriodic("p",7,0,0)` and `strongPeriodic("q",7,1,1)`. In general, requirements can be formed as conjunctions and disjunctions of atomic specifications. Because requirements are composed using a translation to automata, no constraints are imposed on periodicity. Specifically, we do not assume harmonic periods.

5.2 Temporal Logic

Another specification language enabled by our tool, using GOAL, is the safety fragment of linear temporal logic (LTL) [11, 22]. Following are some specification patterns written in temporal logic. The full syntax is described in [31].

- `temporalLogic([](p <--> ~q))`: at any logical execution slot, either method `p` or `q` execute, but not both.
- `temporalLogic([]~(p /\ q))`: the methods `p` and `q` should never run in the same logical execution slot.
- `temporalLogic([](p --> <-> q))`: run `p` only if `q` is invoked in the preceding slot.
- `temporalLogic([](p --> q U r))`: if `p` is invoked then `q` must run in all slots until `r` runs.

Temporal logic is an alternative to drawing automata, often allowing shorter and more intuitive representation.

5.3 Exponential stability

Control software is a specific type of real-time components where the high level requirements are expressed in terms of performance statistics such as exponential stability (see e.g. the case-study described in Section 2, above).

The method `Translator.stability` is an implementation of the algorithm given in [32] for translating exponential stability requirements to automata:

- `stability({`A_1`,`A_2`},{"p","~p"},10,.5)` : for square matrices A_1 and A_2, assuming the dynamics $x(t+1) = A_{\sigma(t)}x(t)$ where

$$\sigma(t) = \begin{cases} 1 & \text{if } \texttt{p} \text{ runs at the } t\text{th slot;} \\ 2 & \text{otherwise,} \end{cases}$$

  assure that, for all t, $|x(t+10)/x(t)| < .5$ no matter what $x(t)$ is.

This type of specification is useful when the assignment of tasks to slot affect the dynamics of a physical system, as follows.

Assume, for example, that a real-time component is designed to stabilize a physical plant. Let $x(1), x(2), \ldots$ be the states of the system at the beginning of each logical execution slot. In many cases, the evolution of x can be described by the linear switched system $x(t+1) = A_{\sigma(t)}x(t)$ where $\sigma(t)$ is a function of the tasks assigned to the tth slot, modeled by predicates as demonstrated above. Then, the method `Translator.stability` can be used to translate stability requirements to scheduling specifications.

We propose the following recipe for combining our tool with tools such as Simulink [28]:

1. Design a sampled-data linear switched system with modes that require excessive computations and other modes that can be implemented with light computations.
2. Extract the state-space matrices for the modes (using e.g. `linearize` built-in MATLAB script).
3. Generate code for each mode (manually or automatically). Wrap the implementation of each mode by a method of a class called e.g. `Controller`.
4. Use `Translator.stability` to generate an automaton for class `Controller`.

6. EXPERIMENTS FOR DYNAMIC CONTROL PERFORMANCE

To illustrate different scenarios in which performance of real-time systems can benefit from the flexibility offered by RTComposer, we describe some illustrative experiments. The experiments involve applications of the system described in Section 2, consisting of two software controllers implemented as real-time components.

For the experiments, software simulations of the controlled plants are implemented. The simulations run as high priority interrupt handlers that update the state of the plant, at high speed, according to its state-space model. These tasks run independently of RTComposer, interacting with the controllers using shared memory.

The controllers are design as Linear Quadratic Regulators (LQR), as described in [2]. The scheduling specification automata are computed using the `Translator.stability` function, described in the preceding section. All reported data is collected on a machine with AMD Athlon 64 Dual Core processor running SUN™ Java RTS 2.1 on SUSE™ Linux Enterprise Real Time 10 operating system.

6.1 Varying load

As a first experiment, we study the ability of RTComposer to adjust to varying load conditions. Particularly, referring to Figure 7, assume that the amount of background computations vary in time and that we want the macro-scheduler to adjust to this factor.

Specifically, assume that a parameter $\gamma \in [0,1]$ is available to the scheduler that represents the current load in the system ($\gamma = 1$ means high background load, $\gamma = 0$ means no background computations). We want to schedule heavier computations when γ is low and lighter computations when γ is high, but always assuring the minimal performance requirements (i.e. always choosing a schedule within the language of the product automaton). Note that real-time threads have priority over background computations, so it is up to the macro-scheduler to schedule less real-time computations when the background load is high.

To achieve this requirement, the following execution mechanism is used. As said earlier, a random walk is simulated over the product of all the specifications, but the probabilities of choosing successors are not necessarily uniform. Specifically, assume that we are in a state with m successors $s_1, \ldots, s_m$. For each $i = 1, \ldots, m$, let t_i be the sum of worst case execution times of the functions that we need to execute if we choose to step to s_i, divided by the slot length (i.e. t_i is the fraction of the logical execution slot that is going to be used if s_i is chosen to be the next state). Then,the probability $p(i)$ of choosing s_i as the next state is given by

$$p(i) = \frac{\gamma\,(1-t_i)}{\sum_{j=1}^{m}(1-t_j)} + \frac{(1-\gamma)t_i}{\sum_{j=1}^{m} t_j}.$$

Namely, if $\gamma = 1$, the probability is $(1-t_i)/\sum_{j=1}^{m}(1-t_j)$ which means that light computations are more probable. If $\gamma = 0$, the probability is $t_i/\sum_{j=1}^{m} t_j$ which means that heavy computations are more probable. And, if $0 < \gamma < 1$, a convex combination of the two probabilities is used.

This scheduling approach allows dynamic adaptation to time-varying conditions as the macro-scheduler continues its random-walk simulation with varying probabilities, depending on the varying values of γ.

The graphs in Figure 11 demonstrate the gain in performance obtained by applying the ability of RTComposer to dynamically allocate resources based on varying conditions. Specifically, it is apparent that lower values of γ allow better control performance.

6.2 Platform portability

As a second experiment, we study the benefits of using RTComposer for implementing platform portability of real-time applications. Particularly, we experiment with the ability of RTComposer to allow better schedules when a faster platform is used.

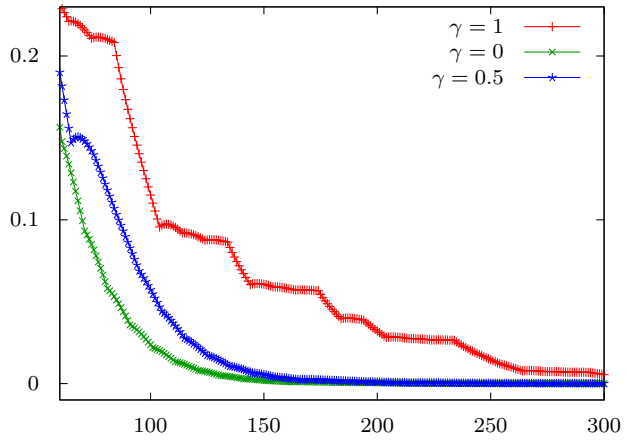

Figure 11: Plot of the state of the controlled system for different values of γ.

Assume, for example, that the system runs on a platform that executes all functions twice as fast as the reference platform. In that case, the heavy computations of both the controllers can execute in the same execution slot. In particular, this allows the macro-schedule depicted in Figure 12.

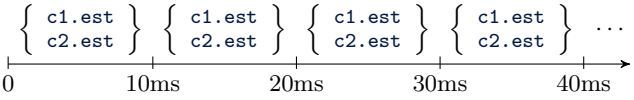

Figure 12: A macro-schedule allowed on a fast platform.

Compared to the schedule depicted in Figure 2, this schedule takes advantage of the better platform to deliver best performance for both control loops.

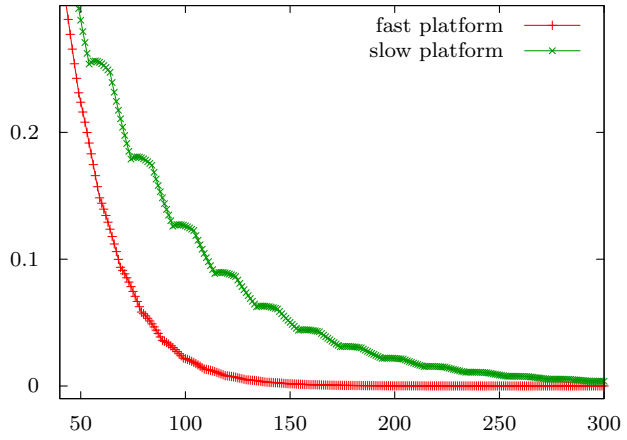

Figure 13: Performance of the controlled loop with different platforms.

The gain in performance is displayed in Figure 13. Clearly, when running on the faster platform, RTComposer chooses a better schedule which is translated to better controller

performance. Note that platform independence can be implemented also without macro-scheduling (by choosing long enough logical execution slots), but the unique feature of our approach is that the system performs better with the faster platform (without re-engineering).

6.3 Dynamic admission control

The last experiment we describe shows how dynamic admission control allows the scheduler to allocate resources based on current need and not necessarily as it would allocate them when all components are loaded.

As discussed in Section 4.4, RTComposer supports dynamic admission control. A typical use for this mechanism is when a real-time software is designed for configurable hardware. In this case, one may want to install and remove software components when the corresponding hardware is installed or removes, respectively.

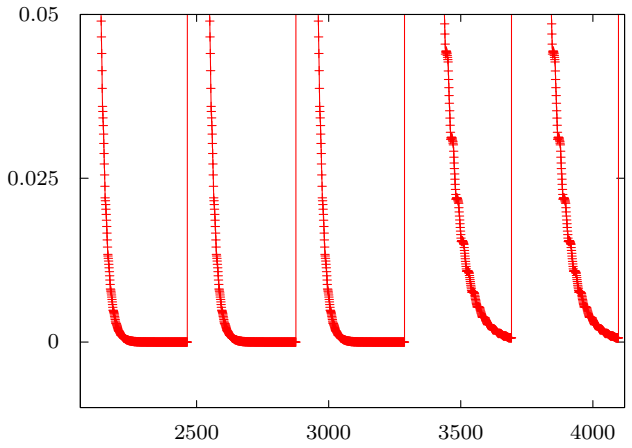

Figure 14: Response to periodic disturbance degrades when a new component is added (at time 3300).

Figure 14 depicts the response of the control loop to a periodic disturbance. It is apparent that the response until time 3300 is better than the response after this time. This degradation of performance is due to admission of a second component that only allows the controller to run estimation every other slot. The computation of the new automaton, which is done as a background computation, takes about half a minute.

This experiment demonstrates the ability of RTComposer to deliver guaranteed performance while using temporarily free resources to improve performance, as opposed to applying a constant schedule that guarantees performance at the price of worst-case performance all the time.

7. RELATED WORK

Many researchers have identified the lack of composability as a problem for scalable component-based design and integration, and offered composable and hierarchical scheduling frameworks based on the classical periodic task model [10, 23, 24, 27]. For example, [27] proposes the periodic resource model, where the specification of a component consists of (T, C) meaning that the component should get C units of computation every T units of time, and shows how to abstract a set of periodic tasks with EDF or rate-monotonic scheduling policies into a single periodic resource. While these efforts address composability, the expressiveness is still limited to specifying periods for individual components. Formal methods literature consists of general frameworks such as I/O automata [21], fair transition systems [22], and interface automata [8, 9, 29] for capturing interfaces with well-developed theories of composition and refinement. Our use of automata is consistent with such general frameworks, and can be viewed as "light-weight" instantiation for the specific purpose of scheduling. The idea of using formal languages and Büchi automata as an interface to capture the set of acceptable schedules over the alphabet of task identifiers, was first advocated in our recent work [2, 32], which shows how to specify stability of control systems using automata and how such specifications can be applied for the LQG control designs.

The Giotto framework provides an abstract programmer's model for the implementation of real-time software consisting of periodic tasks and mode-switching logic [13, 14]. To abstract away from the effects to low-level scheduling decisions, we use the LET semantics for micro-scheduling as advocated by the Giotto framework. However, our framework explicitly allows dependence on macro-scheduling decisions, without sacrificing the potential for rigorous and modular analysis of possible behaviors of a component.

The Exotasks project offers a Java-based programming environment for developing real-time components [4]. The system ensures that the behavior is time-deterministic even in presence of other Java threads, and the technical focus is on issues such as memory isolation and fast garbage collection. Our focus on composable scheduling specifications is orthogonal, and we wish to explore if the ideas of the two projects can be integrated in a fruitful way.

Another related work is tools for simulating and analysing co-design of control and scheduling [7]. TrueTime is a Simulink based tool for simulating real-time software that execute controller tasks. Jitterbug is a MATLAB-based analysis software that relates quadratic performance to delays and jitters induced by software implementation. Extending these tool to support automata-based scheduling can be useful to support the methodology discussed in this paper.

8. CONCLUSIONS AND FUTURE WORK

The tool RTComposer addresses the need of real-time developers to deliver software components that can be safely integrated with other applications. Particularly, the tool provides means for developing real-time applications in Java, a programming language commonly used for multi-platform integrated applications.

The tool is based on formal specification of scheduling constraints with automata. Specifically, each component is equipped with an automaton specifying which sequences of executions of its methods allows it to maintain required performance. As established in this paper, this interface allows a tool that schedules components in a way that adapt to dynamic conditions such as varying load, platform capabilities and components configuration.

The project page of RTComposer is at `http://www.seas.upenn.edu/~gera/RTComposer` where the implementation, examples and further information can be downloaded.

Acknowledgments

This research was partially supported by NSF grants CSR-EHS 0509143, CNS 0524059, and CPA 0541149.

9. REFERENCES

[1] R. Alur, A. Kanade, and G. Weiss. Ranking automata and games for prioritized requirements. In *Proceedings of the 20th conference on Computer Aided Verification*, 240–253, 2008.

[2] R. Alur and G. Weiss. Regular specifications of resource requirements for embedded control software. In *Proceedings of the 14th IEEE Real-time and Embedded Technology and Applications*, 2008.

[3] M. Anand, S. Fischmeister, and I. Lee. Composition techniques for tree communication schedules. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, 235–246, 2007.

[4] J. Auerbach, D. F. Bacon, D. T. Iercan, C. M. Kirsch, V. T. Rajan, H. Roeck, and R. Trummer. Java takes flight: time-portable real-time programming with exotasks. In *Proceedings of the conference on Languages, Compilers, and Tools for Embedded Systems*, 51–62, 2007.

[5] G. Bollella and J. Gosling. The real-time specification for java. *IEEE Computer*, 33(6):47–54, 2000.

[6] G. Buttazo. *Hard real-time computing systems: Predictable scheduling algorithms and applications.* Kluwer Academic Publishers, 1997.

[7] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzén. How does control timing affect performance? *IEEE Control Systems Magazine*, 23(3):16–30, June 2003.

[8] A. Chakrabarti, L. de Alfaro, T. Henzinger, and M. Stoelinga. Resource interfaces. In *Embedded Software, 3rd International Conference*, 117–133, 2003.

[9] L. de Alfaro and T. Henzinger. Interface automata. In *Proceedings of the 9th Annual Symposium on Foundations of Software Engineering*, 109–120. ACM Press, 2001.

[10] Z. Deng and J. Liu. Scheduling real-time applications in an open environment. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, 308–319, 1997.

[11] E. A. Emerson. Alternative semantics for temporal logics. *Theor. Comput. Sci.*, 26:121–130, 1983.

[12] E. Farcas, C. Farcas, W. Pree, and J. Templ. Transparent distribution of real-time components based on logical execution time. In *Proceedings of the conference on Languages, Compilers, and Tools for Embedded Systems*, 31–39, 2005.

[13] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[14] T. Henzinger and C. Kirsch. The embedded machine: Predictable, portable, real-time code. In *Proceedings of the Conference on Programming Language Design and Implementation*, 315–326, 2002.

[15] T. Henzinger and J. Sifakis. The embedded systems design challenge. In *Proceedings of the 14th International Symposium on Formal Methods*, 1–15, 2006.

[16] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[17] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, 2000.

[18] H. Kopetz and G. Bauer. The time triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[19] E. Lee. What's ahead for embedded software. *IEEE Computer*, 18–26, September 2000.

[20] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[21] N. Lynch, R. Segala, and F. Vaandrager. Hybrid I/O automata. *Information and Computation*, 185(1):105–157, 2003.

[22] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: Specification.* Springer-Verlag, 1991.

[23] A. Mok and A. Feng. Towards compositionality in real-time resource partitioning based on regularity bounds. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, 129–138, 2001.

[24] J. Regehr and J. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd Real-Time Systems Symposium*, 3–14, 2001.

[25] A. Sangiovanni-Vincetelli, L. Carloni, F. D. Bernardinis, and M. Sgori. Benefits and challenges for platform-based design. In *Proceedings of the 41th ACM Design Automation Conference*, 409–414, 2004.

[26] S. Sastry, J. Sztipanovits, R. Bajcsy, and H. Gill. Modeling and design of embedded software. *Proceedings of the IEEE*, 91(1), 2003.

[27] I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *Trans. on Embedded Computing Sys.*, 7(3):1–39, 2008.

[28] `http://www.mathworks.com/products/simulink/`.

[29] L. Thiele, E. Wanderer, and N. Stoimenov. Real-time interfaces for composing real-time systems. In *Proceedings of the 6th International Conference on Embedded Software*, 34–43, 2006.

[30] W. Thomas. Automata on infinite objects. In *Handbook of theoretical computer science, Vol. B*, 133–191. Elsevier, Amsterdam, 1990.

[31] Y.-K. Tsay, Y.-F. Chen, M.-H. Tsai, K.-N. Wu, and W.-C. Chan. Goal: A graphical tool for manipulating büchi automata and temporal formulae. In *Proceedings of the 13th conference on Tools and Algorithms for Construction and Analysis of Systems*, 466–471, 2007.

[32] G. Weiss and R. Alur. Automata based interfaces for control and scheduling. In *Proceedings of the 10th workshop on Hybrid Systems: Computation and Control*, 2007.