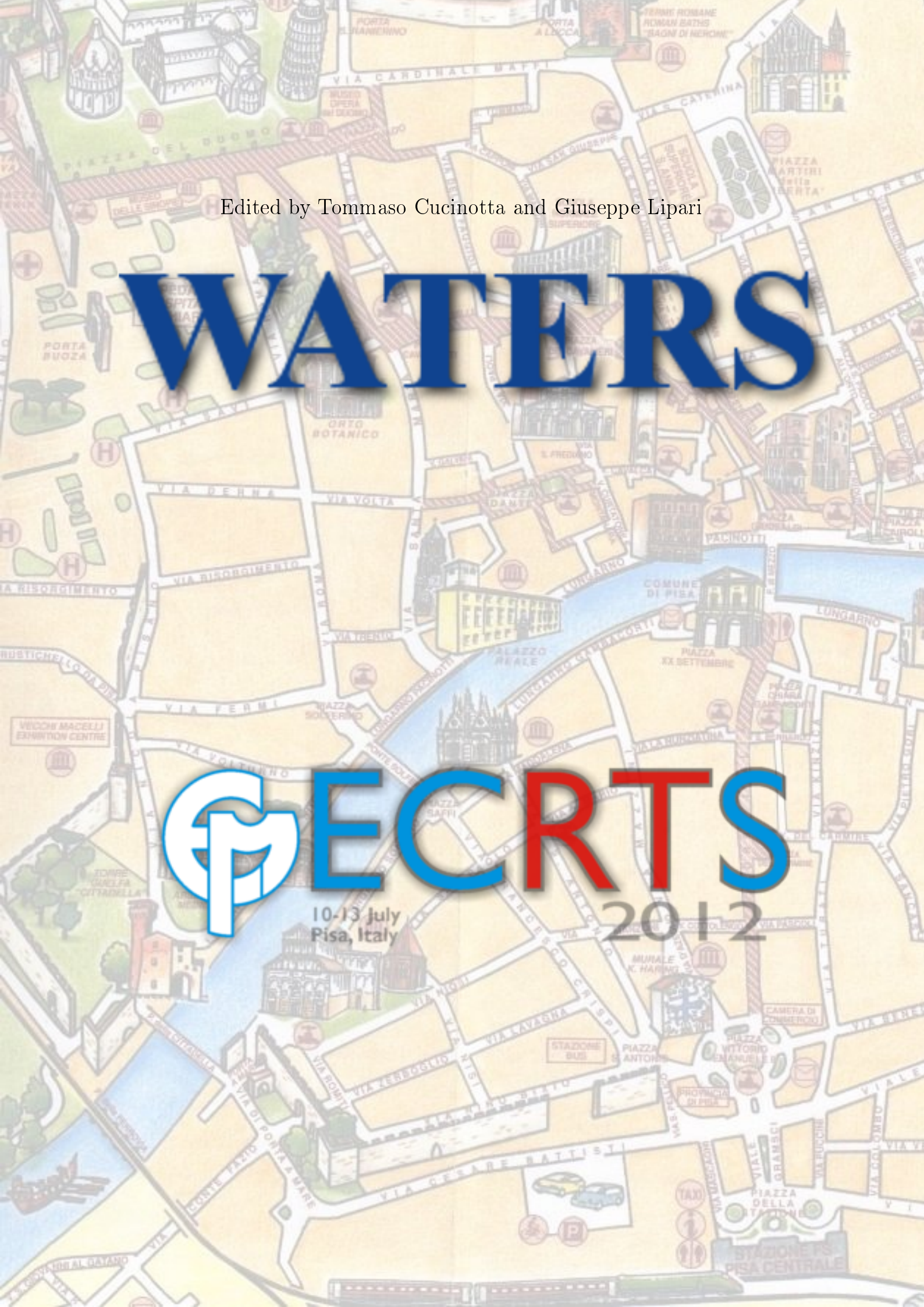
3^{rd} International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems

July 10th, 2012, Pisa, Italy

In conjunction with:
The 24th Euromicro Conference on Real-Time Systems
(ECRTS 2012)
July 10-13, 2012

Edited by Tommaso Cucinotta and Giuseppe Lipari

WATERS



This page has been left intentionally blank.

Table of Contents

Message from the Program Chairs 6
Program Committee 8

Session A: Simulation 1

hsSim: an Extensible Interoperable Object-Oriented n-Level Hierarchical Scheduling Simulator 9
João Pedro Craveiro, Rui Ormonde Silveira and José Rufino

RTMultiSim: A versatile simulator for multiprocessor real-time systems 15
Anca Hangan and Gheorghe Sebestyen

YARTISS: A Tool to Visualize, Test, Compare and Evaluate Real-Time Scheduling Algorithms 21
Younes Chandarli, Frédéric Fauberteau, Damien Masson, Serge Midonnet and Manar Qamhieh

Session B: Design and Analysis

Compositional Performance Analysis in Python with pyCPA 27
Jonas Diemer, Philip Axer and Rolf Ernst

Interoperable Tracing Tools 33
Luca Abeni and Nicola Manica

Advances in the automation of model driven software engineering for hard real-time systems with Ada and the UML Profile for MARTE 39
Julio Medina and Alejandro Perez Ruiz

Enabling Model-Based Development of Distributed Embedded Systems on Open Source and Free Tools 45
Marco Di Natale, Mario Bambagini, Matteo Morelli, Alessandro Passaro, Dario Di Stefano and Giuseppe Arturi

Session C: Simulation 2

SystemC based Simulation for Virtual Prototyping of Large Scale Distributed Embedded Control Systems 51
David Ginsberg, Alessandro Mignogna, Marco Carloni, Francesco Menichelli, Alberto Ferrari, Dang Nguyen and Eelco Scholte

The Design and Implementation of a Simulator for Switched Ethernet Networks 57
Mohammad Ashjaei, Moris Behnam and Thomas Nolte

SoOSiM: Operating System and Programming Language Exploration 63
Christiaan Baaij and Jan Kuper

Message from the Program Chairs

Research in real-time systems has gone very far from the initial seminal papers back in the 70s. Many algorithms, design methodologies, techniques and tools have been proposed, spanning several application areas, from RTOS to distributed systems, from safety critical to soft real-time systems. However, unlike other research areas (e.g., networking) there are no widely recognized reference tools or methodologies for comparing different research works in the area.

Thus, different authors use different algorithms for generating random task sets, different application traces when simulating dynamic real-time systems, different simulation engines when simulating scheduling algorithms. Instead, research in the field of real-time and embedded systems would greatly benefit from the availability of well-engineered, possibly open tools, simulation frameworks and data sets which may constitute a common metrics for evaluating simulation or experimental results in the area. Also, it would be nice to have a possibly wide set of reusable data sets or behavioural models coming from realistic industrial use-cases over which to evaluate the performance of novel algorithms. Availability of such items would increase the possibility to compare novel techniques in dealing with problems already tackled by others from the multifaceted viewpoints of effectiveness, overhead, performance, applicability, etc.

One of the reasons for such a lack of tools is the fact that scientists get little recognition for the software they write. As Michael Nielsen points out in his recent book "Reinventing Discovery: The New Era of Networked Science":

> Today, scientists who write and release code often get little recognition for their work. Someone who has created a terrific open source software program that's used by thousands of other scientists is likely to get little credit from peers. "It's just software" is the response many scientists have to such work. From a career point of view, the author of the code would have been better off spending their time writing a few minor papers that no one reads. This is crazy: a lot of scientific knowledge is far better expressed as code than in the form of a scientific paper.

When we had the idea of this workshop more than three years ago, our goals were to recognise the work of those scientists who write software that is useful for our community; to make these tools more widely known in our community; and to create a small group of researchers interested in contributing with new software and tools.

This is the third edition of our workshop. This year the workshop features 10 papers divided into 3 sessions: Session A and Session C about

simulation of real-time and embedded systems; Session B about design and analysis tools. We also asked the authors to provide their software (or links to their web page where the software is made available). All this information is available through the workshop web page: `http://retis.sssup.it/waters2012/`. Also, a mailing list in Google Group has been set up to distribute information on the workshop themes (`https://groups.google.com/forum/?fromgroups#!forum/ecrts-waters`).

We would like to thank the Euromicro organization for having allowed us to organize this event, and particularly Gerhard Fohler for his prompt and ready support. We would like to thank all the authors for having submitted their work to the workshop for selection, the Program Committee members for their effort in reviewing the papers, the presenters for ensuring interesting sessions, and the attendees for participating into this event. We hope that interesting ideas and discussions will come out of the presentations, demos and the questions that will alternate along the day. We hope you will find this day interesting and enjoyable.

The WATERS 2012 Chairs

Tommaso Cucinotta[1] and Giuseppe Lipari[2]

[1] Tommaso Cucinotta is with Bell Laboratories, Alcatel-Lucent, Dublin, Ireland
e-mail: `tommaso.cucinotta@alcatel-lucent.com`

[2] Giuseppe Lipari is with Laboratoire Spécification et Vérification, École Normal Supérieure de Cachan, France
e-mail: `giuseppe.lipari@lsv.ens-cachan.fr`

Program Committee

- Luca Abeni (University of Trento, Italy)
- Iain Bate (University of York, UK)
- Laura Carnivali (University of Florence, Italy)
- Marco Di Natale (Scuola Superiore Sant'Anna, Pisa, Italy)
- Laurent George (INRIA Paris-Rocquencourt, France)
- Michael Gonzalez (Universidad de Cantabria, Spain)
- Gernot Heiser (NICTA, Australia)
- Mike Holenderski (Eindhoven University of Technology, The Netherlands)
- Thomas Nolte (Mälardalen University, Sweden)
- Stefan Petters (CISTER-ISEP, Porto, Portugal)
- Luca Santinelli (INRIA, Villers-Les-Nancy, France)
- Simon Schliecker (Symtavision GmbH, Braunschweig, Germany)
- Wang Yi (Uppsala University, Sweden)

hsSim: an Extensible Interoperable Object-Oriented n-Level Hierarchical Scheduling Simulator

João Pedro Craveiro, Rui Ormonde Silveira, and José Rufino
Universidade de Lisboa, Faculdade de Ciências, LaSIGE
Lisbon, Portugal
Email: jcraveiro@lasige.di.fc.ul.pt, rsilveira@lasige.di.fc.ul.pt, ruf@di.fc.ul.pt

***Abstract*—Hierarchical scheduling is a recent real-time scheduling topic. It is used to obtain temporal interference isolation in various scenarios, such as scheduling soft real-time aperiodic tasks along with hard real-time periodic tasks, as in mixed-criticality scenarios. Most theory and practice focuses on two-level hierarchies, with a root (global) scheduler managing resource contention by partitions (or scheduling servers), and a local scheduler in each partition/server to schedule the respective tasks. In this paper we describe the development of hsSim, an object-oriented hierarchical scheduling simulator supporting an arbitrary number of levels. With the goal of openness, extensibility and interoperability in mind, due care was put into the design, applying known design patterns where deemed advantageous. We demonstrate hsSim's interoperability potential with a case study involving the Grasp trace visualization toolset.**

I. Introduction

Hierarchical scheduling is a recent topic in the mature real-time scheduling theory discipline. In one of the first works on the topic, Abeni and Buttazo [1] present a hierarchical scheduling framework in which hard real-time tasks are scheduled along with a Constant Bandwidth Server (which, for the scheduler below, behaves exactly has the other periodic tasks); the CBS is, in turn, responsible for scheduling the jobs of aperiodic soft real-time tasks. Hierarchical scheduling is also used in mixed-criticality systems — systems providing multiple functionalities who differ *(i)* in how important ("critical") they are for the overall welfare of the system, and *(ii)* in the level of assurance of each one's mandatory certification [2]. A common approach to this certification issue of mixed-criticality systems is enforcing isolation through time and space partitioning (TSP) [3], such as in the ARINC 653 specification: tasks are separated into partitions, which are scheduled by a cyclic executive (according to a partition scheduling table — PST); in each partition's activity time windows, the respective tasks compete for scheduling based on a priority-based scheduling policy [4], [5]. Finally, hierarchical scheduling also sees application in the virtualization field, with nested virtualization for advanced security purposes evidencing the need to support an arbitrary number of hierarchy levels [6].

This work was developed in the followup of European Space Agency ITI project AIR-II (ARINC 653 In Space RTOS, http://air.di.fc.ul.pt). This work was partially supported by the EC, through project IST-FP7-STREP-288195 (KARYON, http://www.karyon-project.eu/), and by FCT, through the Multiannual and CMU|Portugal programs, and the Individual Doctoral Grant SFRH/BD/60193/2009. This work was partially supported by FCT/Égide (PESSOA programme), through the transnational cooperation project SAPIENT (Scheduling Analysis Principles and Tool for Time- and Space-Partitioned Systems, http://www.navigators.di.fc.ul.pt/wiki/Project:SAPIENT).

In this paper, we present the development of **hsSim** (pronounced *aitch-ess-sim*), a simulation tool for the hierarchical real-time scheduling research community. Due to this target community, we focus on supporting well-known and widely used models therein, such as the periodic task model and, in a near future, compositional analysis abstractions such as the periodic resource model [7]. hsSim pursues the goal of an open, reusable, extensible and interoperable tool. As such, we strived for a modular design and development methodology, aided by the careful application of the object-oriented paradigm and software design patterns. Design patterns are design decisions and solutions for recurring problems encountered in object-oriented systems. The employment of these design patterns caters to the goal of reusing successful past experience in software design [8]. To demonstrate the interoperability potential of hsSim, we describe a case study using Grasp, a trace visualization toolset [9].

Paper outline: In Section II, we present related work on scheduling simulation, including Grasp. In Section III, we briefly describe the system model we assume for the first iterations of hsSim's development. In Section IV, we detail the analysis and design steps taken, with emphasis on the main design patterns from whose application we took advantage. Section V describes the implementation and tests to the current version of hsSim, namely the interoperability case study based on the Grasp toolset. Finally, Section VI closes the paper with concluding remarks and ongoing work.

II. Related work

To the best of our knowledge, the current state of the art lacks a scheduling simulator for hierarchical scheduling frameworks with an arbitrary number of levels.

Cheddar [10] provides a set of scheduling algorithms and policies on which it is capable of performing feasibility tests and/or scheduling simulations for either uniprocessor or multiprocessor environments. In its latest versions, Cheddar already supports schedulability analysis of ARINC 653-like time- and space-partitioned (TSP) systems to some extent [10]. However, Cheddar presents some limitations in its current support thereto. A partition scheduling table (PST) is defined as an array of durations; besides presenting less usability, the current implementation limits the PST to having only one time window per partition per hyperperiod. Cheddar is designed to be extensible, and we are currently involved in a collaboration with the Cheddar team towards more complete hierarchical scheduling

capabilities, applying, among others, the principles and patterns identified in the present paper [11], [12].

Grasp [9] is a trace visualization toolset. It supports the visualization of multiprocessor hierarchical scheduling traces. Traces are recorded from the target system (by the Grasp Recorder or any other appropriate means) into Grasp's own script-like format, and displayed graphically by the Grasp Player. The Grasp toolset does not support simulation and supports only a two-level hierarchy, whereas hsSim simulates hierarchical systems with an arbitrary number of levels. However, the Grasp Player reads traces in a simple text-based format which can be recorded by other tools. In this paper, we demonstrate our tool's interoperability features by implementing the possibility to generate a Grasp trace.

CARTS [13] is an opensource compositional analysis tool for real-time systems, which automatically generates the components' resource interfaces; it does not perform simulation. CARTS relies strongly upon some of the authors' theoretical results (e.g., [14]) and, while implemented in Java, does not take advantage of the latter's object-oriented characteristics (inheritance, polymorphism, encapsulation — especially regarding the separation between domain and user interface). This makes it difficult to be extended and, as such, we chose to develop hsSim from scratch instead of modifying CARTS.

SPARTS [15] is a real-time scheduling simulation focused on power-aware scheduling algorithms. Its simulation engine is optimized by replacing cycle-step execution for an event-driven approach centered on the intervals between consecutive job releases. Hierarchical scheduling support is not mentioned.

MAST 2 [16] defines a model to describe the timing behaviour of real-time systems designed to be analyzable via schedulability analysis techniques; it is accompanied by a tool suite including schedulability analysis tools. MAST 2 introduces modelling elements for virtual resources, abstracting entities that rely on the resource reservation paradigm.

Finally, the Schesim [17] scheduling simulator supports hierarchical limited to two-levels. Simulation is based on models of the tasks' implementations, not on task abstractions such as the periodic/sporadic task models, making it useful for direct application, but not so for our target real-time scheduling theory research.

Regarding commercial tools, a mention is due to SymTA/S [1], a model-based timing analysis and optimization solution with support to ARINC 653. No specific mention is made to hierarchical scheduling, thus we can only assume it supports a two-level hierarchy. Due to its proprietary nature, we cannot fully assert its capabilities and it does not serve our purpose for open, reusable, extensible tools for academic/scientific research.

III. System model

In our model, tasks and partitions are both abstracted as synchronous periodic schedulable entities, characterized by the periodic task model — $\tau_i = \langle C_i, T_i, D_i \rangle$, respectively worst-case execution time (WCET), period, and relative deadline. Each task or partition generates a potentially infinite sequence

[1] https://symtavision.com/symtas.html

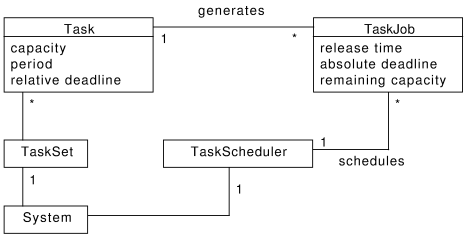


Fig. 1. Traditional 1-level system domain model

of jobs (or activations) characterized as $J_{i,k} = \langle c'_{i,k}, r_{i,k}, d_{i,k} \rangle$, respectively remaining execution time, release time and absolute deadline time; we assume all tasks are released at the critical instant, i.e., $t = 0$. Sporadic tasks are not specifically addressed, but in our model we may abstract them as periodic tasks with periods identical to their minimum inter-arrival times.

The system and the partitions have, each, their own scheduler to schedule their respective children schedulable entities. In traditional two-level hierarchical scheduling systems, such as those defined by the ARINC 653 specification [4], the system scheduler schedules partitions, whereas each partition's scheduler schedules the former's tasks. Here, we aim for a more generic model, where more levels can be composed (i.e., a partition can have children partitions, which in turn have tasks) [7], [14] and tasks can be coscheduled alongside partitions by the same scheduler [1] — hence the need to abstract tasks and partitions.

Finally, we assume scheduling over one unit-speed processor and that task/partition context switching times are either negligible or already accounted for in their temporal characteristics.

IV. Object-Oriented Analysis and Design (OOAD)

The implementation of a tool which we want to be flexible, extensible and more easily maintainable must be preceded by careful analysis and design. We will now document the main analysis and design steps and decisions taken, supported by Unified Modelling Language (UML) diagrams where deemed necessary. The overall design classes diagram is ommited due to paper length constraints.

A. Domain analysis

The traditional real-time system is flat (a 1-level hierarchy). The UML diagram for such a system's domain is pictured in Fig. 1. The system has a flat task set and a task scheduler.

A two-level hierarchical scheduling such as those corresponding to TSP systems [5] can be modelled as seen in Fig. 2. The system has a set of partitions and a root scheduler coordinating which partition is active at each instant. Each partition then has a set of tasks and a local scheduler to schedule the latters' jobs. This domain model strategy has two main drawbacks: it is hard-limited to two levels, and it only allows homogeneous levels (i. e., partitions and tasks cannot coexist at the same level).

B. n-level hierarchy: the Composite pattern

The *Composite* pattern is a design pattern that may be used when there is a need to represent part–whole hierarchies of

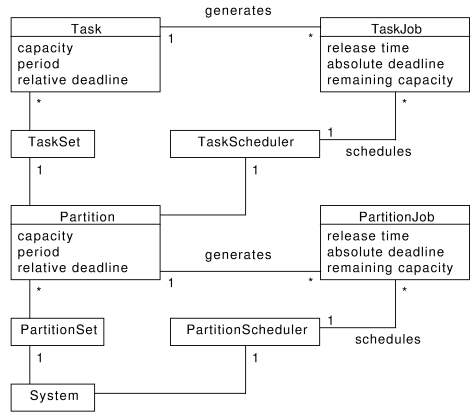


Fig. 2. 2-level hierarchical scheduling system domain model

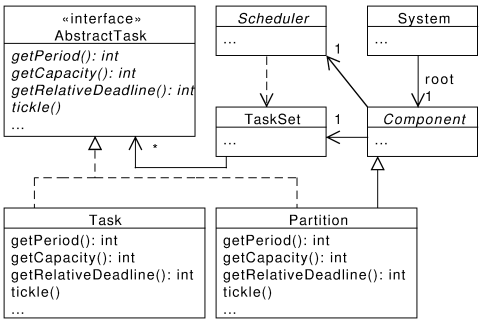


Fig. 3. n-level hierarchical scheduling system using the Composite pattern

objects, and allow clients to ignore the differences between composition of objects and individual objects [8]. Clients manipulate objects in the composition through a component interface, which abstracts individual objects and compositions.

Figure 3 shows the UML representation of the Composite pattern as applied to our domain. Applying this pattern to our model of a hierarchical scheduling framework allows breaking two limitations: the fixed number of levels in the hierarchy [7], [14], and the need for the hierarchy to be balanced [1]. The clients of the AbstractTask interface include schedulers, which will be able to schedule both tasks and partitions through a common interface, reducing implementation efforts and allowing extensions through new schedulable entities (e. g., servers). The application of this pattern triggered further refinements, such as making the TaskSet the System's and Partitions' AbstractTasks container, and merge TaskJobs and partition activations (PartitionJobs) under a generic Job abstraction.

C. Scheduling algorithm encapsulation: the Strategy pattern

The *Strategy* (or *Policy*) pattern is an appropriate solution to when we want to define a family of algorithms which should be interchangeable from the point of view of their clients [8]. In designing hsSim, we apply the Strategy pattern to encapsulate the different scheduling algorithms, as seen in Fig. 4. In the Scheduler abstract class, although we use a scheduling policy to initialize the JobQueue, we leave the obtention of

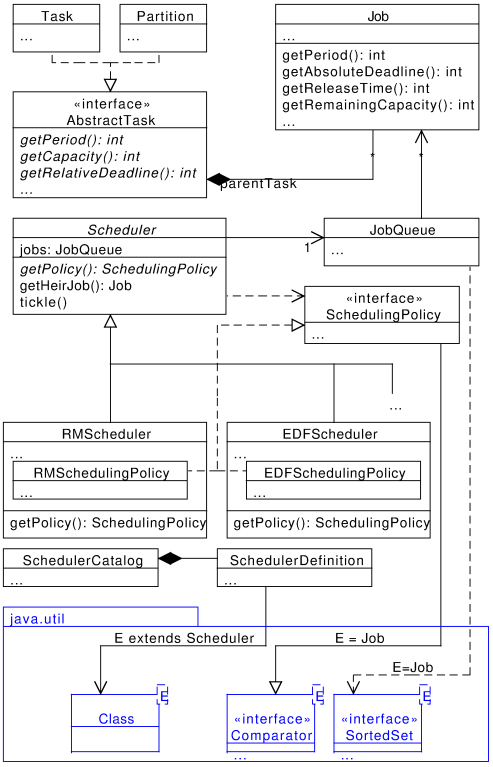


Fig. 4. Scheduling algorithm encapsulation with the Strategy pattern

the scheduling policy (the getPolicy() method) to the concrete scheduler classes; this is supported on another well-known design pattern, the Template Method [8].

The SchedulingPolicy interface extends Java's Comparator interface; this way, an instance of a subclass of SchedulingPolicy can be used to maintain the scheduler's job queue ordered in the manner appropriate for the scheduling algorithm being implemented.

The available strategies (scheduler types) are stored in a catalog, and more strategies can be loaded in runtime (provided the user interface gives a means to it). This is made possible by Java's native reflection capabilities.[2]

D. n-level hierarchy and polymorphism

Due to the design decisions regarding the Composite and Strategy patterns, most operations can implemented without having to account for which scheduler (or schedulers) are present, or for the structure and/or size of the hierarchy (partitions and tasks). Taking advantage of subtype polymorphism, we can invoke methods on Scheduler and AbstractTask references instances without knowing of which specific subtype thereof the instances are.

Let us see how this works with the scheduler tickle operation, which simulates the advance of system execution by one time unit. For the time being, we implement hsSim as a cycle-step execution simulator; an event-driven approach as the one seen

[2]Since we anticipated using the Java to implement hsSim, this and the following design decisions take explicit advantage from facilities provided by the Java libraries.

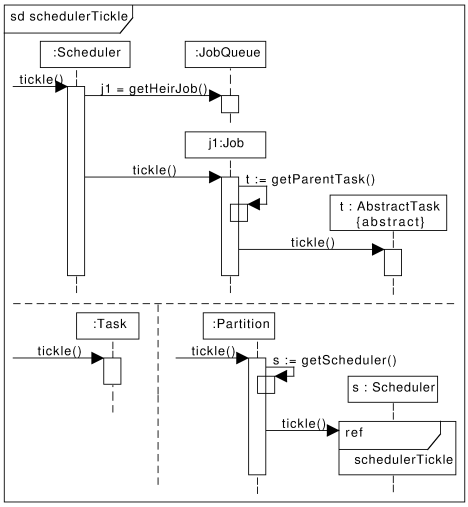


Fig. 5. Sequence diagram for the scheduler tickle operation

in SPARTS [15] is planned for future work (see Section VI). The UML sequence diagram modeling the interactions between objects in this operation is shown in Fig. 5. The hierarchical tickle process is started by invoking the tickle operation on the root scheduler without specific regard for what subtype of Scheduler it is; the right job to execute will be obtained because the scheduler's job queue is maintained accordingly ordered by an instance of an unknown SchedulingPolicy subtype. This job is then tickled, and in turns tickles its parent AbstractTask without knowing if it is a Task or a Partition. It is the job's parent's responsibility to invoke the right behaviour according to its type. If it is a Partition instance, this involves tickling its scheduler; this will cause an identical chain of polymorphic invocations to take place.

E. Decoupling the simulation from the simulated domain using the Observer and Visitor patterns

In hsSim, we want to decouple the simulation aspects (such as running the simulation and logging its occurrences) from the simulated domain itself. On the one hand, we want changes in the simulated domain (a system with partitions, tasks, jobs) to be externally known of, namely by one or more loggers, without the domain objects making specific assumptions about these loggers behaviour or interfaces. On the other hand, we want to be able to create new loggers without tightly coupling them to the domain objects or having to modify the latter. We found the Observer and Visitor patterns to be most appropriate to solve this specific problem.

The *Observer* pattern defines a publisher–subscriber dependency between objects, so that observers (subscribers) are notified automatically of the state changes of the subjects they have subscribed to. The subjects only have to disseminate their state changes to a vaguely known set of observers, in a way that is totally independent of how many observers there are and who they are — in the form of events. The *Visitor* pattern defines a way to represent an operation to be performed on an object hierarchy independently from the latter [8]. In few words, the

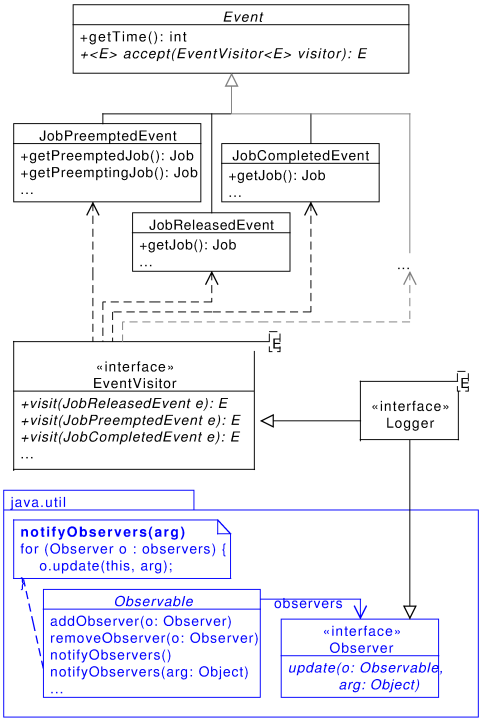


Fig. 6. Logger (Observer and Visitor patterns)

Observer pattern guides loggers in choosing from what domain objects they wish to receive events, and the Visitor pattern helps each logger define what to do with each kind of event.

Our application of these patterns in hsSim is pictured in Fig. 6. We take advantage from the simple Observer implementation provided by Java, with the Logger interface extending the Observer interface — thus obligating its subclasses' instances (the concrete loggers) to provide the method to be called to notify it of an event. The Logger interface also extends our EventVisitor interface, which defines methods to process each type of event. The visit methods are overloaded and every Event subclass provides the following method:

```
public <E> E accept(EventVisitor<E> visitor) {
    visitor.visit(this);
}
```

This way, when receiving an event e, a logger only has to invoke `((Event) e).accept(this)` to have the right visit method called.

We also apply the Observer pattern to establish a dependency between the system clock and the schedulers, so that the latter become aware of when to released their tasks' jobs.

V. Implementation and Use

For a first approach, we deemed necessary one scheduler for each of Carpenter et al.'s priority-based categories [18]: Rate Monotonic (RM) for fixed task priority, Earliest Deadline First (EDF) for fixed job priority, and Least Laxity First (LLF) for dynamic priority. We defined the following iteration strategy to have incrementally more complete prototypes:

1) get the core working with the three chosen schedulers (RM, EDF, LLF) and predefined input/output;

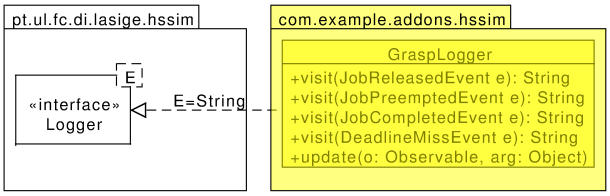


Fig. 7. GraspLogger implementing the Logger interface

Listing 1. GraspLogger excerpt (event notification reception)

```
public void update(Observer o, Object arg) {
    String result = ((Event) arg).accept(this);
    writeToFile(result);
}
```

2) implement a concrete logger extending the Logger interface (Section IV-E);
3) read the simulated scenario from an XML file;
4) implement a simple random scenario generator, with the option of either writing the generated scenario to an XML file or returning a System instance (and respective associated domain objects);
5) implement a text-based user interface for scenario data input and log visualization;
6) implement a graphical user interface for scenario data input, simulation metrics selection, and log visualization.

At the time of this paper's writing, we had completed the first three iterations, implementing both a plain-text logger and a logger which records traces in the format interpreted by the Grasp player [9]. We now describe the latter as a case study for hsSim's interoperability potential. XML reading is left out of this paper due to paper length restrictions.

A. Interoperability: a case study with Grasp

Because of our low coupling design for the event logging (Section IV-E), it is straightforward to trace the simulation to the format interpreted by Grasp Player [9]. The Grasp logger is implemented exactly as if it were developed by an external team, who could even only have access to the core of hsSim as a library. We implement the Logger generic interface, instantiating its type variable with the String type, since we want the processing (visit) of events to return the text to be added to the Grasp trace (Fig. 7).

The visit methods are invoked when an event notification is received, via the update method (Listing 1). Invocation is done indirectly through the accept method, so the right visit method is automatically selected and called.

Along the visit methods, we implement the following mapping between hsSim events and Grasp trace content:

- arrival of Job $J_{i,k}$ (of AbstractTask τ_i)
  1) if τ_i is a Partition, **serverReplenished** $\boldsymbol{\tau_i}$ $\boldsymbol{c'_{i,k}}$
  2) if τ_i is a Task, **jobArrived** $\boldsymbol{J_{i,k}}$ $\boldsymbol{\tau_i}$
- obtention of processor by Job $J_{p,q}$ (of AbstractTask τ_p)[3]
  1) if τ_p is a Partition, **serverResumed** $\boldsymbol{\tau_p}$ (followed by a **jobResumed** log of the task job the partition was last executing)
  2) if τ_p is a Task, **jobResumed** $\boldsymbol{J_{p,q}}$
- preemption of Job $J_{i,k}$ by Job $J_{p,q}$
  1) if τ_i, τ_p are Partitions, **serverPreempted** $\boldsymbol{\tau_i}$ (followed by a **jobPreempted** log of the job the preempted partition was last executing, **serverResumed** $\boldsymbol{\tau_p}$, and a **jobResumed** log of the task job the preempting partition was last executing)
  2) if τ_i, τ_p are Tasks, **jobPreempted** $\boldsymbol{J_{i,k}}$ **-target** $\boldsymbol{J_{p,q}}$ (followed by **jobResumed** $\boldsymbol{J_{p,q}}$)
- completion of Job $J_{i,k}$
  1) if τ_i is a Partition, **serverDepleted** $\boldsymbol{\tau_i}$
  2) if τ_i is a Task, **jobCompleted** $\boldsymbol{J_{i,k}}$

[3]The obtention of the processor from idle by a job is implemented as a JobPreemptedEvent with a null preempted job.

Listing 2. GraspLogger excerpt (visit of a job release event)

```
public String visit(JobReleasedEvent e) {
    Job job = e.getJob();
    StringBuilder sb = new StringBuilder();
    sb.append("plot␣");
    sb.append(Integer.toString(e.getTime());
    sb.append('␣');
    if (job.getParentTask() instanceof Task) {
        sb.append("jobArrived␣");
        sb.append(job.toStringId());
        sb.append('␣');
        sb.append(job.getParentTask().toStringId());
    } else { //Partition
        sb.append("serverReplenished␣");
        sb.append(j.getParentTask().toStringId());
        sb.append('␣');
        sb.append(j.getRemainingCapacity());
    }
    sb.append(System.getProperty("line.separator"));
    return sb.toString();
}
```

In Listing 2, we can see, as an example, the visit method responsible for processing a JobReleasedEvent event.

Although hsSim's core supports an arbitrary number of levels, we now show for the sake of simplicity an example using the current implementation of GraspLogger with a 2-level hierarchical setting.

B. Example test

The Rate Monotonic is used to schedule partitions and to schedules tasks in each partition. This examples is of a sample from the experiments of [19], and the timing characteristics of the partitions are derived from those of the respective tasks by applying Shin and Lee's periodic resource model [7]:

- partition 1 has a capacity of 16 over a period of 75 and contains 4 tasks ($\tau_i = \langle C_i, T_i, D_i \rangle$): $\tau_1 = \langle 46, 500, 500 \rangle$, $\tau_2 = \langle 71, 1000, 1000 \rangle$, $\tau_3 = \langle 25, 1000, 1000 \rangle$, and $\tau_4 = \langle 29, 2000, 2000 \rangle$;
- partition 2 has a capacity of 5 over a period of 25 and contains 2 tasks: $\tau_5 = \langle 32, 250, 250 \rangle$ and $\tau_6 = \langle 67, 1000, 1000 \rangle$;
- partition 3 has a capacity of 5 over a period of 25 and contains 3 tasks: $\tau_7 = \langle 27, 250, 250 \rangle$, $\tau_8 = \langle 109, 2000, 2000 \rangle$, and $\tau_9 = \langle 53, 2000, 2000 \rangle$.

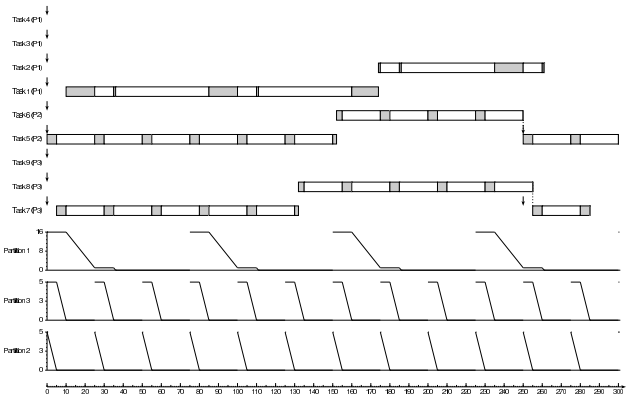


Fig. 8. Grasp Player output excerpt

For the trace generated by hsSim's GraspLogger (omitted due to paper length constraints), Grasp Player displays the graphical output partially pictured in Fig. 8. Partitions are traced as scheduling servers whose budget is consumed while it is active (independently of the execution of children tasks) and replenished at the beginning of each new period. The release, execution, preemption, and finishing of tasks are represented in a Gantt-like chart for each task.

VI. Conclusion and ongoing work

We have described the development of hsSim, an n-level hierarchical scheduling simulator. We emphasized the object-oriented analysis and design decisions, such as the careful application of design patterns, through which we pursued the purpose of yielding an open, reusable, extensible and interoperable tool. Applying the Composite and Strategy patterns allows implementing system operations independently from, respectively, the hierarchy's structure and size and the underlying scheduling algorithms; furthermore, the application of the Observer and Visitor patterns allows great flexibility to add new and diverse simulation loggers, since the simulation and logging aspects are decoupled from the domain concepts. We demonstrate this extensibility and interoperability features with a logger to trace the simulation to the format interpreted by Grasp Player.

Development continues after this submission, so more advances are expected in time for the WATERS 2012 event. Ongoing work includes opensourcing hsSim's code[4], and implementing more types of schedulers (namely those corresponding to scheduling servers, non-preemptive schedulers, cyclic executive and global/partitioned multiprocessor schedulers). Regarding multiprocessor support, much of these features' design (supporting homogeneous and heterogeneous multiprocessor platforms) is already done, but omitted from this paper. Further work planned for after supporting multiprocessor includes adding the option to recursively compute a partition's timing requirements from those of its children using compositional analysis [7], [20], [21] and paying further attention to the specificities of dealing with aperiodic/sporadic tasks.

We have an ongoing collaboration with the Cheddar team, and as such some of the advances made in the development of hsSim should be ported there to enhance its support to hierarchical scheduling and compositional analysis [10]–[12].

[4]http://code.google.com/p/hssim/

Acknowledgment

The authors would like to thank our SAPIENT project partners from Lab-STICC/UBO (Brest, France), especially Frank Singhoff, for useful discussions which helped improve the work presented in this paper.

References

[1] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *RTSS '98*, Madrid, Spain, 1998, pp. 4–13.

[2] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie, "Scheduling real-time mixed-criticality jobs," *IEEE Trans. Comput.*, 2011, to appear.

[3] J. Windsor and K. Hjortnaes, "Time and space partitioning in spacecraft avionics," in *SMC-IT 2009*, Jul. 2009, pp. 13–20.

[4] AEEC, "Avionics application software standard interface, part 1 - required services," Aeronautical Radio, Inc., ARINC Spec. 653P1-2, Mar. 2006.

[5] J. Rufino, J. Craveiro, and P. Verissimo, "Architecting robustness and timeliness in a new generation of aerospace systems," in *Architecting Dependable Systems VII*, ser. LNCS, A. Casimiro, R. de Lemos, and C. Gacek, Eds. Springer, Nov. 2010, vol. 6420, pp. 146–170.

[6] B. Kauer, P. Verissimo, and A. Bessani, "Recursive virtual machines for advanced security mechanisms," in *1st Int. Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments (DCDV 2011)*, Hong Kong, Jun. 2011.

[7] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *RTSS '03*, Cancun, Mexico, Dec. 2003.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1997.

[9] M. Holenderski, R. J. Bril, and J. J. Lukkien, "Grasp: Visualizing the behavior of hierarchical multiprocessor real-time systems," in *WATERS 2011*, Porto, Portugal, Jul. 2011.

[10] F. Singhoff, A. Plantec, P. Dissaux, and J. Legrand, "Investigating the usability of real-time scheduling theory with the Cheddar project," *Real-Time Syst.*, vol. 43, no. 3, pp. 259–295, Nov. 2009.

[11] J. Craveiro, J. Rufino, and F. Singhoff, "Architecture, mechanisms and scheduling analysis tool for multicore time- and space-partitioned systems," *ACM SIGBED Rev.*, vol. 8, no. 3, pp. 23–27, Sep. 2011, special issue of ECRTS 2011 WiP session, Porto, Portugal, July 2011.

[12] V. Gaudel, F. Singhoff, A. Plantec, S. Rubini, P. Dissaux, and J. Legrand, "An Ada design pattern recognition tool for AADL performance analysis," in *SIGAda '11*, Denver, CO, Nov. 2011, pp. 61–68.

[13] L. T. X. Phan, J. Lee, A. Easwaran, V. Ramaswamy, I. Lee, and O. Sokolsky, "CARTS: A tool for compositional analysis of real-time systems," in *3rd Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS 2010)*, San Diego, CA, Nov. 2010.

[14] A. Easwaran, M. Anand, and I. Lee, "Compositional analysis framework using EDP resource models," in *RTSS '07*, Tucson, AZ, Dec. 2007.

[15] B. Nikolic, M. A. Awan, and S. M. Petters, "SPARTS: Simulator for Power Aware and Real-Time Systems," in *ICESS 2011*, Changsha, China, Nov. 2011.

[16] M. Gonzalez Harbour, J. J. Gutierrez Garcia, J. M. Drake Moyano, P. López Martínez, and J. C. Palencia Gutierrez, "Modeling distributed real-time systems with MAST 2," *J. Syst. Architect.*, 2012.

[17] Y. Matsubara, Y. Sano, S. Honda, and H. Takada, "An open-source flexible scheduling simulator for real-time applications," in *ISORC 2012*, Shenzhen, China, Apr. 2012.

[18] J. Carpenter, S. Funk, P. Holman, J. Anderson, and S. Baruah, "A categorization of real-time multiprocessor scheduling problems and algorithms," in *Handbook on Scheduling: Algorithms, Methods, and Models*, J. Y.-T. Leung, Ed. Chapman & Hall/CRC, 2004.

[19] J. P. Craveiro, J. Rosa, and J. Rufino, "Towards self-adaptive scheduling in time- and space-partitioned systems," in *RTSS 2011: WiP Session*, Vienna, Austria, Nov./Dec. 2011.

[20] I. Shin, A. Easwaran, and I. Lee, "Hierarchical scheduling framework for virtual clustering of multiprocessors," in *ECRTS 2008*, Prague, Czech Rep., Jul. 2008.

[21] J. P. Craveiro and J. Rufino, "Heterogeneous multiprocessor compositional real-time scheduling," in *RTSOPS 2012*, Pisa, Italy, Jul. 2012.

RTMultiSim: A Versatile Simulator for Multiprocessor Real-Time Systems

Anca Hangan, Gheorghe Sebestyen
Computer Science Department
Technical University of Cluj-Napoca
Romania
{Anca.Hangan, Gheorghe.Sebestyen}@cs.utcluj.ro

Abstract— **This paper presents a simulation tool that can be used for the evaluation of real-time behavior for a wide range of parallel and distributed systems. The simulator is based on a flexible workload and CPU/execution model that covers different multiprocessor scenarios, from parallel systems and independent task sets to distributed environments and task sequences (transactions). The simulator may be used to verify the real-time schedulability (RT feasibility) of a task set on a given executing environment, to measure time parameters of tasks during execution or to find a feasible task allocation scenario (setup) for a given set of globally defined parameters.**

Keywords: real-time systems; simulation; multiprocessor; clustered scheduling; task set generation;

I. INTRODUCTION

As multiprocessor systems are becoming the execution environment for most of today's real-time applications, developers need theoretical methods and experimental tools for evaluating the feasibility and time behavior of such systems. As showed in many recent papers [1,2], real-time analysis on multiprocessor systems is not a trivial task, and in the general case when different restrictions (synchronization, casual dependencies, data race conditions, etc.) are considered, beside the time conditions, the problem has a NP complexity.

In order to handle this complexity in a more pragmatic way we developed a methodology that combines theoretical feasibility results [3,4] with a simulation process. In this way, we can evaluate the real-time behavior of a given set of tasks executed on a multiprocessor execution environment.

Our goal was to develop a real-time simulator that can cover a multitude of cases from parallel architectures to distributed ones and from independent task sets to transaction of tasks or fork-join parallel threads. We also included aspects of network communication.

The resulting tool called *RTMultiSim* is a discrete time simulator that can be used to measure the time parameters of such systems and in predefined scenarios to demonstrate the feasibility of a real-time scheduling policy. It can be useful in evaluating the statistical influence of different parameters (e.g. CPU utilization, parallelism degree) over the real-time behavior of multiprocessor system. In addition, we developed a task set generation tool, which provides input for the simulation tool.

The simulation tool is useful in system modeling and design phases in order to establish the number of required processors, the maximum utilization/load factor, the worst-case response time of critical tasks, or to demonstrate the feasibility of a given setup.

A. Related Work

In the field of real-time system's analysis and simulation, there are a number of solutions and tools, offered as open source software (e.g. MAST, STORM) or as commercial products. The first ones are more generic, treating the real-time behavior of systems at higher abstraction levels; the commercial ones (e.g Simulink) are more related to some pragmatic solutions or specific platforms.

The differences between these tools are regarding the following aspects:

- The workload model used in simulation – types of executing units (tasks, threads), periodicity of jobs, processor affinity, clustering, etc.
- The execution environment model – uniprocessor, parallel or distributed systems, uniform or heterogeneous execution, with or without communication (networking).
- The execution and scheduling model – fixed or dynamic priorities, time-triggered or event-triggered execution.
- The real-time parameters and restrictions model – discrete time, global or local time.
- Non-real-time conditions accepted in the model – task synchronization, causal dependencies, and concurrent access to common resources.

In our approach, we tried to cover as many scenarios as possible, allowing seamless variation between different models. The above-mentioned model types may be obtained as particular cases through the tool's parametric configuration. This is not the case for a number of existing simulation tools, specialized for a given workload and system model.

The closest tool to our approach is STORM (Simulation tool for real-time multiprocessor scheduling) [5]. STORM can handle multiprocessor architectures and data exchange between periodic tasks. The simulated system's description is specified through an XML file. Compared to our tool, this simulator does not cover intra task parallelism (e.g. multi threading or fork-join model). It uses only global EDF scheduling strategy and it does not allow partitioned and clustered scheduling approaches.

MAST (Modeling and analysis suite for real-time applications) [6] allows the analysis of uniprocessor, multiprocessor and distributed systems as well. It uses an event-triggered execution model with the possibility to handle complex task dependencies. As scheduling strategy, a fixed priority approach is used.

FORTAS (Framework for real-time analysis and simulation) [7] is a real-time system analyzer and simulator. It offers functionalities for feasibility testing of various multiprocessor scheduling algorithms as well as for viewing task schedules. It also provides a task set generation tool.

Compared to MAST and FORTAS, which are focused on analytical scheduling evaluation, our approach is mainly concentrated on simulation-based evaluation offering step-by-step information about the evolution of the system under test. Our tool implements a clustered scheduling approach and it allows intra-task parallelism. Moreover, in the same simulated system, different scheduling strategies can be defined for each particular executing element (processor, or network).

B. *Paper organization*

The remainder of the paper is organized as follows: Section II presents the simulation environment and its concepts. Simulation execution is described in section III. Section IV introduces our approach to task set generation. Simulation examples are presented in section V. Section VI concludes the paper.

II. THE SIMULATION ENVIRONMENT

The *RTMultiSim* simulation environment is defined through the specification of a workload, an execution environment (platform) and a scheduling strategy. This information, obtained from user-defined input parameters, is transformed into components of the simulator as shown in Fig.1. The simulator executes the task set on the available CPUs according to the selected scheduling strategy. During the simulation process, it records relevant time parameters related with the behavior of the system.

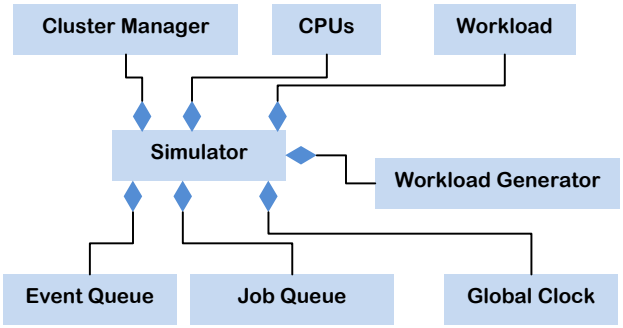


Figure 1. The simulation environment

A. *The Workload Model*

The main concepts, which define the workload model, are showed in Fig. 2: transactions, tasks, messages, events, execution sections and synchronization points.

Transactions are sequences of tasks and messages executed periodically and which are represented through directed acyclic graphs (DAGs). Graph edges represent precedence dependencies. A real-time transaction has the following defining elements:

- *Task graph* (*G*) – a DAG that describes the dependencies between tasks and messages and determines the execution order
- *Period* (*T*) – the repetition period of a transaction.
- *Deadline* (*D*) – the deadline of a transaction, relative to its release time.

A *task* in our model has the following parameters:

- Phase (ϕ) – the release time of the first job (job=execution instance of a task).
- Period (*T*) – task's repetition period.
- Deadline (*d*) – time limit relative to the release of the job.
- A sequence of *execution segments*.
- Consumed and produced events.
- CPU affinity – list of processors on which the task can be executed.

The *message* is a special type of "task", which is handled by a node representing a network segment.

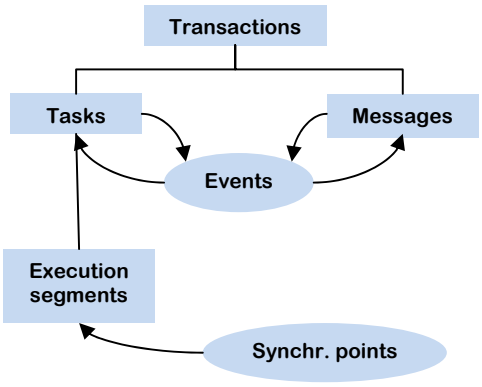


Figure 2. Workload model concepts

Dependencies between tasks or between tasks and messages are implemented using the producer-consumer model. A task (or message) can produce *events* which are consumed by other tasks (or messages), hence creating an execution dependency between producer and consumer. Events produced by executing jobs are stored in the *global event queue*. Consumer jobs extract the expected events from the global queue.

A task is composed of one or more *execution segments*. A segment may be a sequential portion of a task or one that can be executed in parallel on a number of processors. An execution segment has an *execution time* and a *parallelism* value. Between two execution segments, there is a *synchronization point*. An execution segment will begin only if

the previous execution segment is completed. At the beginning of each execution segment, a number of parallel threads equal to the parallelism value will be created. Each thread will have the execution time of the segment to which it belongs. If a task contains only one execution section with a parallelism value equal to 1, its jobs will be single threaded (sequential).

Task phases can be equal to zero or can be chosen at random from an interval. Tasks have their own predefined periods when being part of independent task sets. If tasks are part of transactions, the period is not specified because, by default, it is equal with the transaction's period.

The workload model used by *RTMultiSim* allows the representation of a variety of real-time tasks such as independent sequential periodic tasks, dependent tasks, parallel tasks, fork-join tasks and transactions.

B. The Platform and Scheduling Models

The platform is modeled as a sum of identical processing units (CPUs). The execution speed of all tasks is the same on all processors. Delays caused by memory transfers are not considered. Job context switch time and job migrations can be taken into consideration as constant values.

The scheduler implements a *clustered* approach. Processors are grouped into clusters. In clustered scheduling, job migration is restricted to a subset of processors. Each cluster has its own scheduler and job queue. First, tasks are allocated to clusters, and then each cluster scheduler globally schedules jobs inside the cluster. It can easily be observed that the global and partitioned scheduling are instances of clustered scheduling. If the cluster size is equal to the total number of processors, then we have *global scheduling* and if each cluster contains only one processor (the number of clusters is equal to the number of processors), we have *partitioned scheduling*.

The *RTMultiSim* scheduling model (Fig. 3) contains a cluster manager and a set of clusters. The cluster manager applies a partitioning heuristic to assign all tasks to the existing clusters.

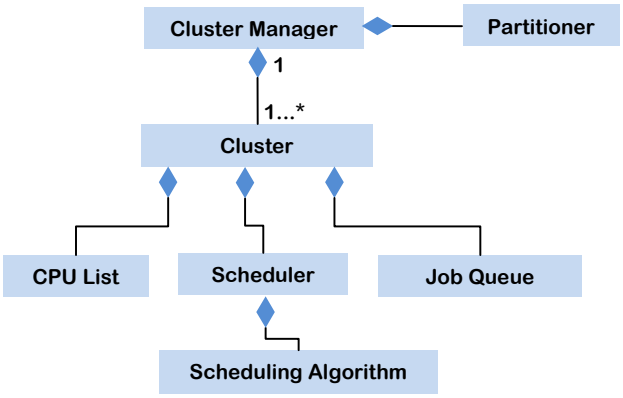


Figure 3. The RTMultiSim scheduling model

A *cluster* contains the assigned task set, the cluster description (list of CPUs), a global job queue and a scheduler. A local scheduling algorithm decides the execution order of jobs. In our model, each cluster can use a different scheduling algorithm.

The partitioning heuristics available in *RTMultiSim* are: *Next Fit* and *Affinity*. In *Next Fit* method tasks are sorted in a decreasing order according to their utilization factor (computation time divided with period). Each time, the first task is assigned to the next available cluster. In *Affinity* method tasks with largest utilization factor and the lowest CPU Affinity number are allocated first. Each task is allocated to the first available cluster, which contains a CPU from its CPU Affinity list. New partitioning heuristics can be added by writing classes, which implement a predefined abstract interface.

The available scheduling algorithms are: *Rate Monotonic* (RM), *Earliest Deadline First* (EDF), *Least Laxity First* (LLF) and *First In First Out* (FIFO). The application may be extended with user-defined scheduling algorithms by creating classes which implement a predefined abstract interface.

III. Simulation Execution

The simulation is performed for a fixed time interval or until the feasibility interval is covered. For instance according to [3,4] the feasibility interval for fixed priority algorithms on multiprocessor systems is a multiple of the task set hyper-period.

A. Simulation Time and Workload Generation

The simulation time is modeled as a *global clock*. All CPUs are synchronized to this clock. The global clock advances with one simulation time unit (STU). At each clock tick (transition), the state of the system is recomputed. There can be new job releases and, depending on the scheduling strategy, there can be schedule updates that generate preemptions. A job's execution time is equal to an integer number of STUs. In a time step, each CPU executes exactly one time unit from the total execution time of a running job.

The *workload generator* creates new jobs according to the workload model. After each global clock transition, this component gets from the workload model all the tasks that have to release new jobs at the current simulation time. Based on those tasks, it creates new jobs, which are copied in the *global job queue*.

B. Jobs and Threads

In *RTMultiSim*, jobs inherit all task parameters. The job release time is set to its creation time. Job execution is performed according to the task's sequence of execution segments.

A job contains a list of *threads*, which can be started in parallel. Threads are created at the start of an execution segment. When all the threads in an execution segment are completed, the threads for the next segment are created. A thread can pass through several states during its existence, from creation to completion. The possible thread states are: *ready*, *scheduled*, *running*, *blocked* and *completed*. In ready state a thread is prepared for scheduling. If it was chosen by the scheduler and assigned to a CPU, the thread is in scheduled state. During execution, the thread is in running state. The thread is blocked if it waits for an event to be produced, in order to start or resume its execution.

C. Scheduling

The cluster manager takes the jobs from the global job queue and places them in the clusters' job queues, according to task partitioning. Then, each cluster's scheduler chooses the jobs that will be executed on each of its CPUs, according to the scheduling algorithm. Threads belonging to the same job have the same priority. Ready threads that belong to the same job will not interrupt running threads. A multithreaded job can be allocated to more than one CPU at the same time.

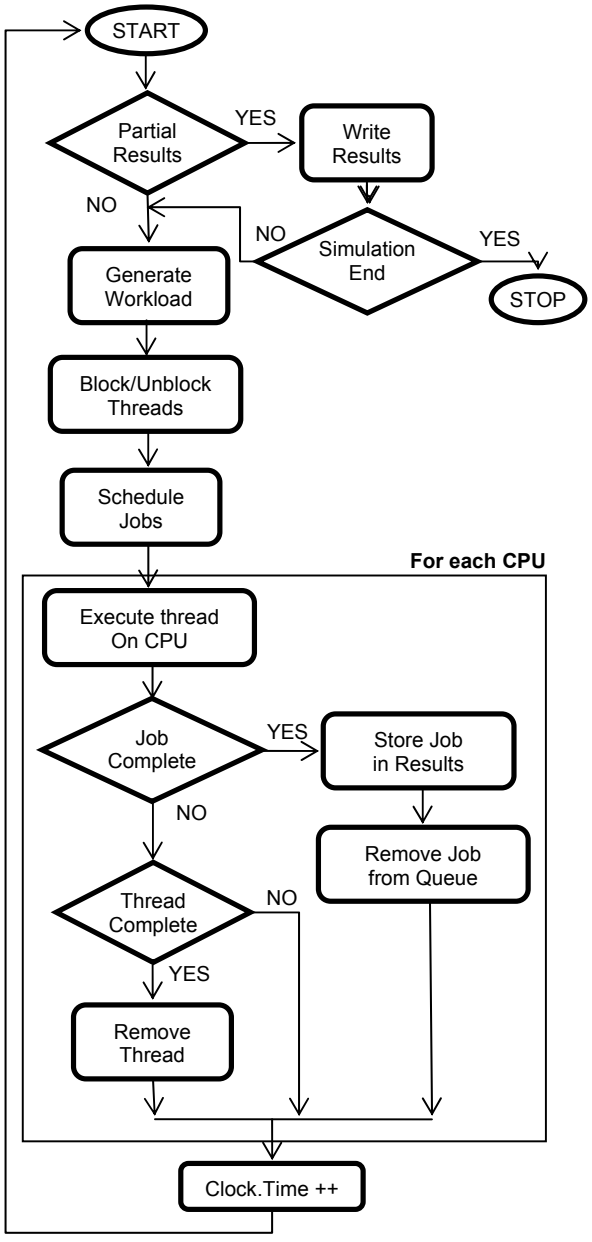


Figure 4. Simulation execution flowchart

D. Execution

A simulation execution step, as showed in Fig. 4, starts with workload generation. Generated threads that do not meet the requirements to enter ready state (e.g. the thread can start only if a certain event is produced) are blocked. Blocked threads that meet the requirements to enter the ready state are unblocked.

New jobs, if any, are placed in the clusters' job queues. Each cluster scheduler allocates jobs (threads) to its CPUs. Next, the threads are executed on CPUs. During thread execution, events can be consumed or produced, execution time is increased and execution statistics are recorded. If the thread is completed, it is removed from the jobs' current threads list. If the job is completed, it is removed from the cluster's queue and placed in the results.

Finally, the simulation time is increased and the simulation execution moves to the next time step.

E. Simulation Results

Simulation results are periodically written in a database, until simulation completion. Results for each simulation are recorded separately. For each released job, the simulator records the response times and deadline. For each thread, release time, start time, completed time, execution time, number of migrations and CPU visitation sequence are recorded. Based on the recorded raw data a number of statistical parameters may be computed such as: the number of successfully finished tasks, number of deadline misses, number of migrations, effective utilization of CPUs, etc.

IV. Task Set Generation

In order to generate different simulation scenarios that fit to some globally defined parameters (e.g. processor usage) we developed a tool for automatic task set generation. The goal is to establish for every simulation scenario some task parameters that cover statistically the most relevant cases.

Task sets used for evaluations are generated automatically in the majority of cases. The method used to generate tasks is essential, as some task set characteristics like the task set cardinality, the distribution of task periods or the distribution of individual task utilizations may influence the evaluation. For instance, for the same task set utilization factor, we obtained different schedulability results when we used uniform and exponential task utilization distribution. For example, on 6 processors, with utilization factor of 4.8 and global EDF scheduling, task sets generated with a uniform distribution were better scheduled then those obtained with an exponential task utilization distribution.

We needed a tool that generates task sets that do not produce misleading simulation results. The main problem to be solved is to generate n individual utilization values of which the sum is equal to U. Even though there are two important results, which address this problem, the UUnifast-Discard algorithm [8] and the Randfixedsum algorithm [9], we decided to use a new approach. We made this decision because, in the UUniFast-Discard case, the algorithm fails to generate task sets for particular values of n and U [9], and because Randfixedsum is very complex and difficult to understand and implement.

To obtain a task set with n independent periodic tasks with implicit deadlines and utilization equal to U, we developed the following methodology:

1. Randomly choose n task periods (T_i) uniformly distributed in the interval [T_{min}, T_{max}], having the least common multiple (*LCM)* less than a given LCM_{max}.
2. Generate n random task utilization values (u_i) for which the sum is equal to a given U. Each u_i has to be equal or greater than $1/T_i$ and less than 1 (because the computed execution time C_i has to be greater than 0).
3. For each pair (T_i, u_i), compute the execution time $C_i=u_i*T_i$ of task i.
4. Verify if the task set satisfies the requested parameters. If not, the task set is discarded.

For step 2, we propose an algorithm that generates n task utilization values with the sum equal to U (see Fig. 5). The algorithm starts with assigning each u_i the mean value (U/n). To obtain a random distribution of the utilization values inside the task set and keep the total utilization equal to U, at each iteration step we randomly choose two u_i values which will be modified by adding and subtracting a random value from the interval [$0,U/n$]. After a large number of iterations, we obtain task utilization values, which are well spread in the interval [$1/T_i$, 1].

Algorithm: generate n utilization values of sum equal to U.
Input: n, U, T_1,T_2,…,T_n
Output: u_1, u_2, …,u_n

```
Begin

For(i=1; i<=n){ ui=U/n;}
Repeat n^4 times
{
 d = random(0,U/n);
 x = random(1,n);
 y = random(1,n);
 if(x!=y)&&(1/Tx<=ux-d<=1)&&(1/Ty<=uy-d<=1)
 {
   ux = ux+d;
   uy = uy-d;
 }
}

End
```

Figure 5. The algorithm which generates n utilization values with the sum equal to *U*.

We compared our results with the results of Randfixedsum and we concluded that the two approaches are equivalent because they generate similar distributions of individual utilization values. In terms of execution time, our algorithm is slightly less efficient, but it produces results in an acceptable time for less than 100 tasks per task set.

V. SIMULATION EXAMPLES

RTMultiSim can simulate a variety of systems, defined through configuration parameters, without any changes in the code. From the platform point of view, these systems can span from multiprocessor (parallel) to distributed architectures. The scheduling strategies may be global, partitioned, or clustered. The workload can be represented as independent parallel and periodic tasks, distributed transactions, dependent periodic tasks and fork-join parallel tasks.

The workload is specified in XML or text files and other simulation parameters such as number of processors, cluster configuration, scheduling and partitioning strategies are set in the RTMultiSim GUI.

An independent task set has the following specification:

```
<Tasks>
  <Task type="periodic" id="1" C="10" T="15" D="12" />
  <Task type="parallel" id="2" C="5,4,6" P="1,3,1" T="15" D="15" />
  …
</Tasks>
```

Where *id* is the task identifier, *C* is the execution time or list of execution times (for each execution segment), *T* is the period, *D* is the deadline, and *P* is a list of parallelism values, one for each execution segment.

A transaction is specified as follows:

```
;60,60
t1 child m1
m1 child t2
t2 child
```

Where the first two values are the period and the deadline *t1*, *t2* are tasks and *m1* is a message. For more details regarding system specification, please refer to the *RTMultiSim* user help.

Until now, we used *RTMultiSim* to conduct simulations in scenarios created for different research objectives, such as:

- Feasibility assessment of a given task set on a specific platform and scheduling strategy.
- Evaluation of a platform and scheduling strategy for workloads with variable parameters (e.g. utilization factor, task set cardinality).
- Assigning time parameters (e.g. intermediate deadlines, periods) for a given task set in order to assure schedulability on a specific platform.

Starting from the theoretical results presented in [3,4] we developed a feasibility test for a given independent synchronous periodic task set on a multiprocessor with global EDF scheduling. The authors of [3,4] demonstrated that for fixed priority algorithms the interval in which the feasibility must be tested (called feasibility interval) is a multiple of the task set hyper-period. If there are no deadline misses in the feasibility interval, and if the system state is the same at the beginning and at the end of the interval, then the task set is feasible. With our simulation tool, we can determine deadline misses and make a comparison between the system states at the two ends of the feasibility interval.

Using the simulator, we evaluated the Rate Monotonic algorithm's efficiency in the global, partitioned, and clustered scheduling approaches, in terms of deadline misses and migration rate [10]. We generated sets of tasks with increasing total utilization factor. On a platform with multiple processors,

we simulated the task sets in all scheduling approaches and compared the results. The experiments showed that a modified clustered solution assures a better result [10].

Our experimental measurements made on different kinds of task set distributions revealed the dependency between schedulability, utilization factors and task set cardinality. Task sets with large cardinalities can be scheduled easier because individual task utilizations are smaller. Fig. 6 shows the experimental results obtained through simulation for 8 processors: the number of schedulable task sets as a function of task set cardinality and task set utilization.

We also analyzed the schedulability of task sets on multiprocessors, scheduled with global EDF. We showed that an important gap (un-decidable region) between theoretically determined necessary and sufficient schedulability conditions, presented in [2], could be overcome through simulation. We applied the schedulability test in [11] on a group of task sets and we conducted simulations in the feasibility interval for the same group of task sets. After comparing the obtained results in the two experiments results (see Fig. 6 and 7), we concluded that the simulation approach finds more schedulable task sets than the analytical one.

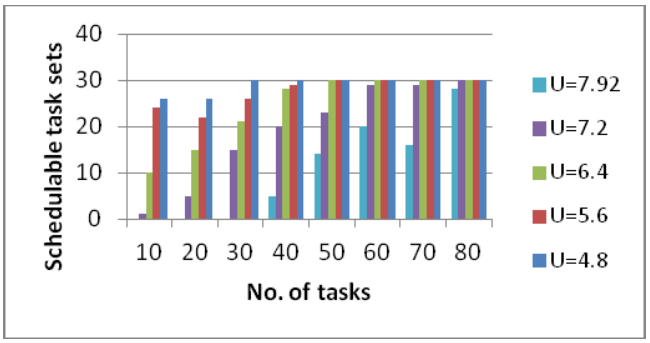


Figure 6. Task set schedulability, function of task set cardinality and task set utilization on 8 processors, obtained through simulation.

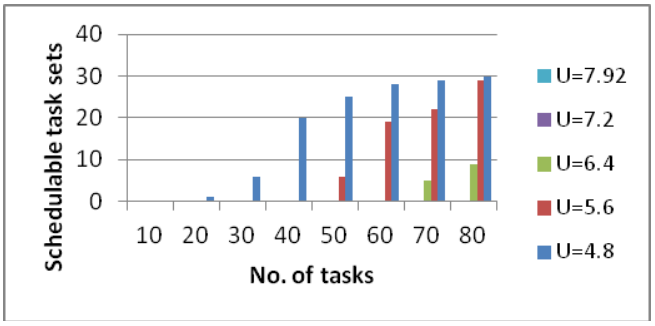


Figure 7. Task set schedulability, function of task set cardinality and task set utilization on 8 processors, obtained with the schedulability test from [11].

In a recent research work, we used the facilities of the simulator together with a genetic algorithm in order to determine suboptimal real-time parameters (e.g. intermediate deadlines) for sequences of tasks and messages grouped in transactions. Using the simulation results, we had the possibility to compute the fitness function of different combinations of task parameters (chromosomes). The result was a task set allocation and deadlines assignment that fulfill the initial real-time requirements.

VI. Conclusions

The paper presents *RTMultiSim*, a versatile simulation tool that covers most of the typical cases of multitasking and multiprocessor real-time systems. First, we described the conceptual model of the simulator and its functioning. The workload model used by the simulator allows representation of various types of real-time tasks such as independent periodic tasks, dependent tasks, parallel tasks, fork-join tasks, and transactions. The scheduling strategy may be global, partitioned, or clustered. *RTMultiSim* provides multiple partitioning and scheduling algorithms and an easy method to integrate new algorithms. We also presented a methodology for automatic task set generation, used to generate different simulation scenarios that fit to some predefined parameters. Finally we gave short descriptions of research work in which the simulation tool proved its usefulness.

As future work, we plan to develop a graphical tool for the representation of simulation results and we intend to implement more partitioning heuristics and scheduling algorithms.

References

[1] R. I. Davis and A. Burns. “A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. ”techreport YCS-2009-443, University of York, Department of Computer Science, 2009.

[2] M. Bertogna, S. Baruah, “Tests for global EDF schedulability analysis”, Journal of Systems Architecture, no. 57, pp. 487–497, 2011.

[3] L. Cucu, J. Goossens, “Feasibility intervals for fixed-priority real-time scheduling on uniform multiprocessors”, ETFA, Prague, September 2006.

[4] L. Cucu-Grosjean, J. Goossens, “Exact schedulability tests for real-time scheduling of periodic tasks on unrelated multiprocessor platforms”, Journal of Systems Architecture, 2011.

[5] R. Urunuela, A. Depaposlanche, and Y. Trinquet, "Storm a simulation tool for real-time multiprocessor scheduling evaluation," in Emerging Technologies & Factory Automation (ETFA), 2010 IEEE Conf., pp. 1-8, Sep 2010.

[6] M. G. Harbour, J. J. G. García, J. C. P. Gutiérrez, and J. M. D. Moyano, “Mast: Modeling and analysis suite for real time applications,” in 13th Euromicro Conference on Real-Time Systems, 2001, p. 125.

[7] P. Courbin, L. George, ”FORTAS : Framework for real-time analysis and simulation”, 2nd International Workshop on Analysis Tools and Methodologies for Embedded and Real-timeSystems, WATERS 2011.

[8] R. I. Davis and A. Burns, “Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems,” in Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS 2009), December 2009, pp. 398–409.

[9] P. Emberson, R. Stafford and R.I Davis, “Techniques for the synthesis of multiprocessor tasksets”, 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-timeSystems, WATERS 2010.

[10] A. Hangan and Gh. Sebestyen, “Simulation-based evaluation of real-time multiprocessor scheduling strategies”, in Proceedings of the 2010 IEEE 6th International Conference on Intelligent Computer Communication and Processing, 2010, pp.375-378.

[11] J. Goossens, S. Funk, S.Baruah, “Priority-driven scheduling of periodic task systems on multiprocessors”, Real Time Systems 25 (2–3) (2001) 187–205.

YARTISS: A Tool to Visualize, Test, Compare and Evaluate Real-Time Scheduling Algorithms

Younès Chandarli*†, Frédéric Fauberteau‡, Damien Masson*, Serge Midonnet† and Manar Qamhieh†
Université Paris-Est, LIGM UMR CNRS 8049,
*ESIEE Paris, 2 bld Blaise Pascal, BP 99, 93162 Noisy-le-Grand CEDEX, France
†Université Paris-Est Marne-la-vallée, 5 bld Descartes, Champs sur Marne, 77454 Marne-la-Vallée Cedex 2, France
‡CEA List, LaSTRE, Point Courrier 94, Gif-sur-Yvette, F-91191 France

***Abstract*—In this paper, we present a free software written in Java, YARTISS, which is a real-time multiprocessor scheduling simulator. It is aimed at comparing user-customized algorithms with ones from the literature on real-time scheduling. This simulator is designed as an easy-to-use modular tool in which new modules can be added without the need to decompress, edit nor recompile existing parts. It can simulate the execution of a large number of concurrent periodic independent task sets on multiprocessor systems and generate clear visual results of the scheduling process (both schedules and tunable metrics presentations). Other task models are already implemented in the simulator, like graph tasks with precedence constraints and it is easily extensible to other task models. Moreover, YARTISS can simulate task sets in which energy consumption is a scheduling parameter in the same manner as Worst Case Execution Time (WCET).**

I. Introduction

In order to evaluate the efficiency of a new approach in real-time systems, software simulation against other algorithms are commonly used. Due to the lack of a standard simulation tool approved by the real-time community, most of the researchers tend to create their own. This situation raises some concerns. On one hand, the presented results are hard to be validated without careful examination of the simulation tool. So these results might be biased toward the proposed approach either by adapted generation of testing tasks or by biased implementation against the compared algorithms. On another hand, reasons as out-of-date simulation tools or lack of good documentation can incite researchers to create new tools, which will lead to repetitive algorithms' implementations specially the common ones (*e.g.* RM, DM, EDF)

while consuming the time and effort of researchers. Moreover, if a standard platform succeeds to emerge, one can compare his own policy with a very complicated one without having to understand the very specificity and optimizations of this one. Finally, the simulation protocols could be standardized, and more easily describable by the use of such a reference tool. In this paper, we introduce YARTISS, a new simulation tool for real-time systems. Genericity is its main feature, by which we hope to overcome the problems mentioned before. New users are allowed to add their own implemented algorithms easily, with no need to understand how the simulator is built or works. We do not pretend to propose a perfect simulator, however we tried during its development to learn from our past tries [1], [2]. YARTISS is written in the Java programming language, which is very popular nowadays and offers valuable attributes regarding portability. In order to ensure independence between the different features of the simulator and to reduce the possibilities of massive failures among them, we used modern programming paradigms, like module oriented programming and Java unit tests (JUnit) oriented development. We tried to develop YARTISS keeping in mind that in order for a simulator to become a reference tool, it should have the following properties: 1) the software must be available under an open source license which gives any researcher the freedom to analyze, verify or modify its implementation ; 2) the *Application Programming Interface* (API) of the software must be well documented and the developer who wants to add or modify an algorithm should not have to read the entire source code in order to understand its behavior ; 3) each part of the simulator (its core, the tasks generator, the results analyzer, ...) must be independent from each other, and easily replaceable by an external module ; and 4) the simulator has to be easy to use in a way that a non-developer researcher can be able to use it. Due to its generality and modularity, we hope that YARTISS makes a valuable contribution to the long process of developing a standard simulation tool recognized by the real-time scheduling research community. We expose our motivations in Section II. We review related works in Section III. Section IV presents the simulator functionalities. The program architecture is described in Section V. Case studies which demonstrate the extensibility of the tool are presented in Section VI. How to get the tool is explained in Section VII. Future works are discussed in Section VIII and finally we conclude in Section IX.

II. Motivations: A Brief History of YARTISS

Our first try in writing a real-time system simulator was called RTSS [1] and developed between 2005 and 2008. The tool was first developed to test some algorithms to handle temporal fault tolerance and was later extended in order to test aperiodic tasks handling algorithms [3], [4]. Lots of modifications had been made in a hurry with some assumptions on the behavior of existing classes without documentation. Then modifying anything could result in errors in another completely different parts. Moreover, although the tool was initially programmed in Java, it began to rely more and more

on bash scripts to be launched and to transform output into human readable files. Based on this first tool, a second one, RTMSim [2], was developed between 2008 and 2011 in the purpose to simulate multiprocessor platforms [5]. The general key ideas were kept, but the first tool had become such complicated and unmaintainable that we had to start it over. Of course, all validated parts of RTSS which were of no interest at the time, were not reimplemented and so were lost (*e.g.* an implementation of D^{OVER} [6]). A third try was made in early 2011, RTSS v2 [1], which was basically a rebuild of RTSS including energy consuming tasks and used for [7]. Unfortunately, even if it is more usable today than the first RTSS, it suffered from the same problems of documentation, modularity and usability to simulate and exploit results of large scale simulations. Moreover, it seems difficult to extend it to simulate multiprocessor platforms. So we came to the development of a new software: YARTISS. From the start, we aimed to produce a tool where the task model, the number of processors and behavior such as the energy consumption model are as easy as possible to modify. Another point on which we focused our attention is the usability of the user interface to produce human readable traces. Our goal was to develop a simulator able to produce evaluations as well as to debug our energy-related algorithms. When we wanted to use YARTISS for an another purpose, namely the simulation of directed graph model of real-time tasks which is a model of tasks with precedence constraints and concurrency (see [8]), this was done without any problems, validating its extensibility.

III. Related Works

There exist a lot of tools to simulate or visualize instrumented real-time systems execution traces. Due to space limitation, we cannot provide here an extensible list of existing tools. For the instrumented execution analyzer tools, one can refer to [9], [10]. Among open simulation tools, we can cite MAST [11], Cheddar [12], STORM [13] and FORTAS [14]. MAST permits modeling distributed real-time systems and offers tools to *e.g.* test their feasibility or perform sensitivity analysis. Cheddar is written in Ada, handles the multiprocessor case and provides many implementations of scheduling, partitioning and analysis algorithms. It also comes with a user-friendly *Graphical User Interface* (GUI). Unfortunately, no API documentation is available to help with the implementation of new algorithms. Moreover, the choice of the Ada language reduces the potential additional developers number. Finally, FORTAS and STORM are tools which, as YARTISS, are written in Java, had modular architectures and permits us to simulate task sets on multiprocessor architectures. They both represent very valuable contributions in the effort to provide open and modular tools and they are good candidates in our opinion to be widely used.

Unfortunately, even if it is more usable in its current state than our previous tools, FORTAS seems to suffer from the same issues: its development is not open to other developers for now, we can only download *.class* files, no documentation is yet provided and it seems that no new version has been released to public since its presentation last year in WATERS.

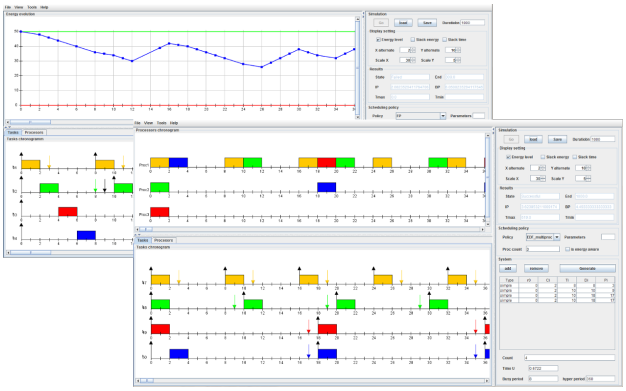


Figure 1. Energy and Multiprocessor Simulation views

IV. Functionalities

The two main features of our simulator are the simulation of the execution of one task set scheduled by a specific scheduling policy and the large-scale comparison of several scheduling policies in different scenarios, which implies its third feature: the random task sets generation.

A. Single Task Set Simulation

Through the GUI, we can load a task set either from a file, by random generation or entering its parameters manually. We can parametrize the desired simulation and run it by the click of a button. Several views are then proposed. The simulation parameters are the task set, the number of processors, the scheduling algorithm and the energy profile.

1) Task Models: YARTISS offers an open architecture that greatly facilitates the integration of different task models. The current version proposes two models, the first one is the Liu and Layland task model augmented with energy related parameters. All the tasks are independent and one task is characterised by its WCET C_i, its worst case energy consumption E_i, its period T_i and its deadline D_i. The second one is the Graph task model which is a common real-time task model on multiprocessor systems. It is used to implement systems consisting of number of missions in which there exist dependencies controlling their execution flow. In this model, a graph G_i is a collection of real-time tasks $\{\tau_{i,1}, \tau_{i,2}, \tau_{i,q}\}$, sharing the same deadline D_i and period P_i of the graph, and they differ in their WCET $C_{i,j}$. The directed edges between the tasks of the graph determine their precedence constraints, and since each task in the graph might have more than one successor and predecessor, concurrent execution can be generated. We will see in Section VI-C that it is easy to propose other task models.

2) Uniprocessor / Multiprocessor: Using the simulator, one can implement and test his own multiprocessor algorithms and partitioning policies. Some multiprocessor scheduling

algorithms were implemented to test this feature like EDF and FP.

3) Energy Profile: Unlike many other simulators, this one permits us to model the production and the consumption of energy in the system. It permits the user to model an energy harvester like a battery or a capacitor with limited or unlimited capacity. It can also model a renewable energy source by a charging function. The user can implement and use his own energy profiles. Figure 1 shows the GUI. Note that the view used to print the energy level can easily be augmented to print other metrics, such as system slack times for example.

a) Energy Source Model: We have implemented an energy source profile that models a renewable energy source represented by a battery with limited capacity and a linear charging function. This model is not the only possible one, the user can add his own profile by implementing the interface and injecting it into the engine of the simulator in few lines of code and without the need to open packages. An example is given in Section VI-B.

b) Consumption Model: It is important to note that for some works, energy consumption of a task must be modeled independently from its WCET [15]. This is why our simulator provides the ability to specify a consumption profile for each task of the system or choose one global profile applied to all tasks. A consumption model is represented by a function and must be able to provide the amount of energy consumed between two dates during the tasks execution i.e. the integral of the consumption function. Implemented models so far are: *Linear consumption* (not realistic but permits establishing some interesting preliminary conclusions) and *Early instantaneous consumption* where all the energy cost of a task is consumed as soon as a task is scheduled. This later model is assumed to represent the worst case scenario. As the energy source profile, a new consumption model can be added without having to open the simulator packages. An example is given in Section VI-B.

4) Scheduling Policy: The main purpose of the simulator is to test scheduling algorithms, compare them and show their performances and efficiency. Much attention has been focused on the design of this part of the simulator to make it as generic as possible so that users can add, override and inject new scheduling policies easily. There are currently twenty algorithms implemented including classic algorithms (RM, DM, EDF uni- and multiprocessor), heuristics for the energy constrained scheduling problem and policies for precedence graph model based on *Least-Laxity-First* (LLF). As with other parameters of the simulator run-time environment, the user can add and link his own algorithms in some lines of code without open core packages. An example is given in Section VI-A.

B. Run Large Scale Simulations

A major utility of the simulator is the large scale comparison of several algorithms or scheduling policies. It is done in the same way of a simple single simulation but on a large set of systems on different scenarios. The comparison is based on statistics that currently can be the number of failures or missed deadlines, the system lifetime, the amount of time spent at maximum energy level E_{max} and minimum level E_{min} and the average duration of idle period and busy periods. One can add his own metrics as demonstrated in Section VI-E. Multiple simulations are run concurrently by using the java multi-threading concept and so the duration of simulations is greatly reduced, taking advantage of hardware parallelism. We show in Figure 2 examples of charts which can be displayed with YARTISS.

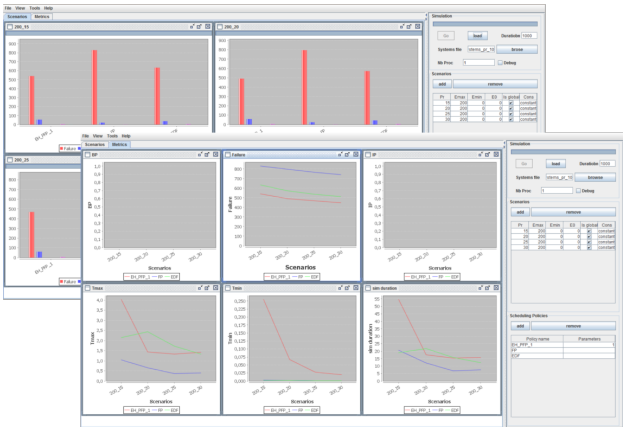


Figure 2. Concurrent large scale simulations: histogram and curves views

C. Task Sets Generation

Performing large-scale tests requires a large set of task systems. To be credible, we have to use sufficiently varied systems to cover the possible task systems space. The simulator provides the ability to choose a generator according to desired scenarios and algorithms. The current version includes a generator inspired by the UUniFast-Discard algorithm [16] adapted to energy constraints. This algorithm generates task sets that respect the CPU utilization ($U = \sum \frac{C_i}{T_i}$) and the energy utilization ($U_e = \sum \frac{E_i}{T_i \times Pr} where Pr is the recharging function$) imposed by the user. The basic version was not energy aware. We had to adapt it to produce time feasible and energy feasible systems. The principle is to distribute the load imposed on the tasks which compose the system. When we add energy cost to the task and an energy load to the system we end up with two parameters to vary and two conditions to satisfy. The algorithm in its current version distribute U and U_e in the same way on the tasks then tries to find the pair (C_i, E_i) which satisfies all the conditions namely U_i, U_e and $Pr < \frac{E_i}{C_i} - Pr < E_{max}$. The operation is repeated a few times and keeps the pair that approaches most the imposed conditions, finally, the algorithm returns a time and potentially[1] energy feasible system. The user can use the described generator as he can write and use his own.

[1]Until now there is no feasibility test that takes into account energy constraints, we hope to have the possibility to present some key ideas to RTSOPS, conjointly organized with ECRTS and WATERS

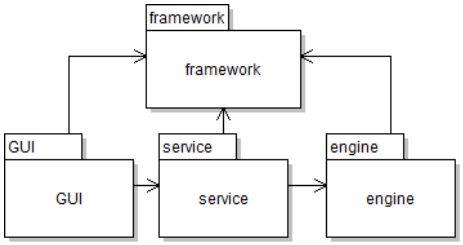


Figure 3. Modules connexion UML Diagram

D. Graphical User Interface

To facilitate the use of the simulator by a large number of users, we provide our application with a GUI to make the features mentioned above available in an interactive and intuitive way. After the simulation of a single system with an energy profile and a scheduling policy, the user can follow and analyze the schedule on three different views: a time chart, a processor view and the energy curve which shows the evolution of energy (as mentioned before, other data can be monitored and print on this view). In order to run simulations and get the results of a comparison of scheduling policies, the application offers a view that allows the user to select the scheduling policies to be compared, the energy scenarios and to run simulations. Thus the user can see the results as one graph per scenario or per comparison criterion. This view offers also a debugging tool in which the user can analyze the result of comparisons system by system and can optionally display the time chart of each system and in each scheduling policy. This can help us to detect behaviors that differ in one algorithm to another. Then this simulator can produce results on a large scale of randomly generated task systems in order to evaluate a scheduling policy, but also easily explore properties of a new algorithm, we can find counter examples on hypothesis by easily isolating degenerate cases. For example, in the case of energy scheduling, no optimal algorithm exists yet. In order to test empirically if a new algorithm is optimal, an approach consists in running it on a large number of task systems, and ask the simulator to present only the systems where this algorithm fails whereas other heuristics succeed.

V. Architecture

To meet the requirements specified in Section II, we have ensured that the design is as generic and open as possible by applying the appropriate design patterns and modular programming practices. We cut the project in four main modules: the engine or core module, the service module responsible for handling input/output with the engine, a module for GUI and finally a framework module that contains general tools necessary for the application. This module separation follows the classical *Model-View-Controller* design pattern (see Figure 3) which permits us to isolate the core application part from its presentation and thus permits the engine to be generic and easily integrable in other tools.

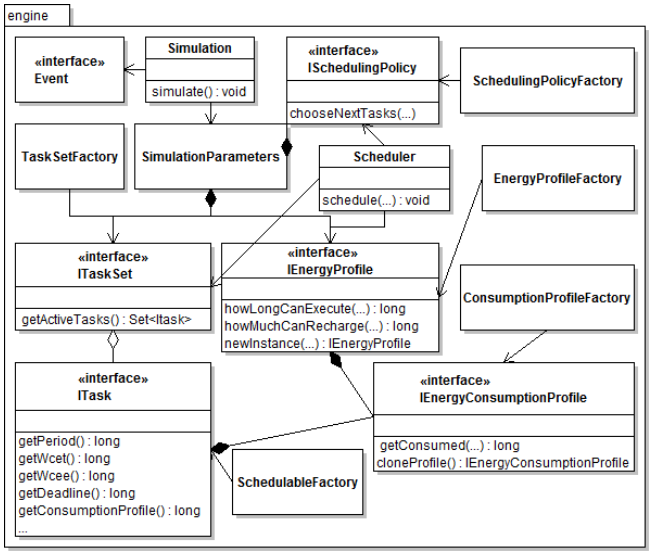


Figure 4. The engine module UML diagram

A. Engine Module

The *Simulation* class, responsible of running a simulation, takes as parameters a container which embeds a scheduling policy, an energy profile and a set of tasks. An UML diagram of this module is given in Figure 4. The simulator is event-triggered: on receipt of an event the scheduler is called to update the running tasks. It then calls the scheduling policy to choose the tasks to execute and the associated processors. This module so defines interfaces needed to execute (*i.e.* the scheduling policy) the energy source model and task energy consumption model. The interface implementation is not directly linked to the simulation object. In order to build an energy profile, for example, one has to register an instance of this class in a factory. This factory is responsible of creating new instances when needed and completely hides the implementation. This allows anybody to create his own scheduling policy or his own task consumption model in a transparent way: one only has to write the model code and register an instance of his class by calling a method of the factory in order to make the new class available through the GUI. Case study are given in Section VI in order to demonstrate this assertion.

B. Service Module

This module makes the interface between the simulator core and the user interfaces. It serves the necessary data to the GUI and gets back the user modifications from it. It also enables to prepare the simulation parameters, or to set up a large scale test. This module component has been made in such a way it permits us to reuse the same classes for an other interface. (*e.g.* a textual user interface).

C. Framework Module

This is a toolbox module that contains generic classes and functions in order to facilitate the code writing. This module

```java
public class MainDemoSP {
  public static void main(String[] args) {
    SchedulingPolicyFactory.registerPolicy(new LLF());
    DesktopMain main  = new DesktopMain();
    main.setVisible(true);
  }
}

class LLF extends AbstractMultiProcSchedulingPolicy{
  @Override public String getPolicyName() {
    return "LLF";
  }
  @Override public ITaskSet createTaskSet() {
    return new AbstractTaskSet(new Comparator<ITask>() {
      @Override  public int compare(ITask t1, ITask t2) {
        long laxity1  = t1.getDeadline() - t1.getRemainingCost();
        long laxity2 = t2.getDeadline() - t2.getRemainingCost();
        int cmp =  (int) (laxity1 - laxity2);
        if(cmp==0)
          return (int) (t1.getPriority() - t2.getPriority());
        return cmp;
      }
    }) {
      @Override  public SortedSet<ITask> getActiveTasks(long date) {
        SortedSet<ITask> activeTasks = new TreeSet<ITask>(comparator);
        for (ITask t : this)
          if (t.isActive())
            activeTasks.add(t);
        return activeTasks;
      }
    };
  }

  @Override  public Processor[] chooseNextTasks(
      Processor[] processors,  ITaskSet taskSet,
      IEnergyProfile energyProfile, long date,
      EventGenerator evGen) {
    int i=0;
    for (ITask task : taskSet.getActiveTasks(date)) {
      if(i<processors.length){
        long hlcet = energyProfile.howLongCanExecute(task);
        if (hlcet <= 0) {
          evGen.generateEvent("energy_failure", task, date, null);
          processors[i].setNextTask(null);
        }
        else {
          evGen.generateEvent("check_energy_state", task, date + 1, null);
          processors[i].setNextTask(task);
        }
      }
      i++;
    }
    for(;i<processors.length; i++){
      processors[i].setNextTask(null);
    }
    return processors;
  }

  @Override  public ISchedulingPolicy newInstance() {
    return new LLF();
  }
}
```

Listing 1. How to add a scheduling policy

makes good use of the Java concurrency API and capacities of modern multiprocessor to accelerate the execution of several simulations. It follows the *producer consumer* design pattern in order to permit *e.g.* to run several simulations in the same time, sending the result of each one to the consumer which computes statistics and updates the GUI. This is also used for the task set generation. Several producers can run in concurrency, sending the produced tasks to consumers which write them in files. This speeds up the generation and simulation of large scale tests.

D. View Module

This module contains the necessary classes for the GUI.

VI. Case Studies

We demonstrate in this section that it is easy to tune the simulator to address specific needs.

A. Adding a Scheduling Policy

To add a new scheduling policy, *e.g.* LLF, one first needs to add the simulator's *.jar* files to a new project in his favorite IDE and then to provide an implementation of the interface *ISchedulingPolicy*. The policies are named to identify them among all others. The method *getPolicyName()* must so return the policy's name. To permit the scheduling policy factory to instantiate the new class, one must implement the method *newInstance()* that returns a new instance of his policy class. Then he specifies the task set model by implementing the method *createTaskSet()*, giving him the choice to use the available task models or to create a new one according to how tasks must be sorted. *ITaskSet* is an iterable of *ITasks* that sorts tasks and returns a sorted set of activated tasks at time t. For LLF, tasks must be sorted by their laxity. To decide which tasks to execute and on which processor the method *chooseNextTasks()* must be implemented. Listing 1 shows the code needed to use LLF as an external module with the simulator.

```java
public class MainDemo {
  public static void main(String[] args) {
    SchedulingPolicyFactory.registerPolicy(new LLF());
    ConsumptionProfileFactory.registerConsumptionProfile(new LogConsumption());
    DesktopMain main  = new DesktopMain();
    main.setVisible(true);
  }
}

class LogConsumption implements IEnergyConsumptionProfile {
  @Override public String getName() {return "log";}
  @Override public List<Double> getParameters() {return null;}
  @Override public void setParameters(List<Double> params) {}

  @Override public long getConsumed(long wcet, long wcee,
                                    long romainingTimeCost,  long duration) {
    double a = wcet - romainingTimeCost;
    double b = a + duration;
    if(b > wcet) b = wcet;
    if( (b-a) <= 0 )return 0;
    long result = (long) Math.log(b/a);
    if(result > wcee )result = wcee;
    return result;
  }

  @Override public IEnergyConsumptionProfile cloneProfile() {
    return new LogConsumption();
  }
}
```

Listing 2. How to add a new energy profile

B. Adding an Energy Profile

The same methodology can be applied to add a new energy consumption profile. Listing 2 shows the code needed to use a logarithmic consumption profile, still as an external module.

C. Adding a Traffic Model

To add a new task model, one has to implement the interface *ITask* and register the class into the *Schedulable* factory. The current version of the interface describes a standard Liu and Layland task. It can be used in its current state to model another kind of tasks like we do with graph tasks and tasks with precedences without open or modify the packages. If it is not sufficient, one can extend it to make it more suitable to his needs. For example, to implement uncertain task model where the tasks execution times are specified into an interval, one can implement the interface *ITask* and modify the behavior of some methods to permit the exceeding of WCET by manipulating the *remainingCost()* method and the conditions of jobs end. Due to space limitation, we cannot give the code here, but it will be added to the demonstration package code suite (see Section VII).

D. Using an External Module to Generate Tasks

One of the advantages of the simulator is that it works with files of standard format like XML. It lets us use external tools if needed to generate tasks by converting the output file with XSL transformation to get an input file understandable by our simulator.

E. Adding More Metrics

If one wants to count the number of preemptions, for example, he has to modify the class *Simulator* to check each tasks begin and end events to detect preemptions and increment a counter in the statistics container. Then, to show the new metric on GUI he must modify the class *Metric* to add the new one and the necessary algorithms to compute maximum, minimum and average. Clearly, this is not a good design. This point is discussed in Section VIII.

VII. Distribution

The project is available from the GForge collaborative development environment hosted at https://svnigm.univ-mlv.fr/projects/yartiss/. This environment provides a subversion server allowing anonymous checkouts, documentation hosting, RSS feeds subscriptions, and public forums. A web page dedicated to YARTISS is also available at http://yartiss.univ-mlv.fr. In addition to a general presentation of the tool, it proposes a demo applet version which allows interested readers to try YARTISS directly from their web browser and an application form to allow anybody to share external modules.

VIII. Future Works

The actual release offers many important and expandable features but the simulator is still under development. Some parts of the project have been made in a hurry which has prevented them to be as clean as they could. For example, the implementation of comparison metrics is strangely coupled with simulation classes and if we want to add a new metric we will be forced to open the engine module and modify internal classes as described in Section VI-E. This may be dangerous and not acceptable architecturally. Improvements are planned to address such weaknesses, like we have done with energy profiles and scheduling policies. Some other future works are planned: 1) We want to provide a command line user interface to allow the use of our simulator without the graphic environment to permit its use inside automated scripts and/or through a distant machine. This should be done easily because of the adopted architecture and responsibilities separation. 2) If we use XML format mostly in all inputs and outputs in order to be able to reuse other external tool functionalities, this feature must be generalized to the simulation results in order to permit their visualization with an external tool (*e.g.* GRASP[10]). 3) An additional work is needed on the description of processors and we need to add the ability to execute on heterogeneous and independent processors in terms of computational power, memory and energy consumption. This could also lead to integrate research on distributed systems. 4) Finally, concerning the energy part, we must integrate the Dynamic Voltage and Frequency Scaling (DVFS) model in order to be compliant with most recent works in this area.

IX. Conclusion

In this paper we presented YARTISS, a real-time multiprocessor scheduling simulator. A consequent effort has been made to make it as extensible as possible. To justify the need for an open and generic tool, we presented the history of YARTISS development. Then we briefly presented existing simulation tools. We have described the three main functionalities of YARTISS: 1) simulate a task set on one or several processors while monitoring the system energy consumption, 2) concurrently simulate a large number of task sets and present the results in a user friendly way that permits us to isolate interesting cases, and 3) randomly generate a large number of task sets. Then, in order to demonstrate the modularity and extensibility of our tool, we presented its architecture and five case studies that show how to add functionalities, in most cases without having to open or modify the project archive. Finally we gave the instructions to test YARTISS and presented some improvement features we will implement. We hope that this software can become a first step toward a widely adopted simulation tool through the real-time scheduling community.

References

[1] D. Masson, "RTSS v1 and v2," https://svnigm.univ-mlv.fr/projects/rtsimulator/.

[2] F. Fauberteau, "RTMSIM," http://rtmsim.triaxx.org/.

[3] D. Masson and S. Midonnet, "Userland Approximate Slack Stealer with Low Time Complexity," in *Proc. of RTNS*, 2008, pp. 29–38.

[4] ——, "Handling non-periodic events in real-time java systems," in *Distributed, Embedded and Real-time Java Systems*, M. T. Higuera-Toledano and A. J. Wellings, Eds. Springer US, 2012, pp. 45–77.

[5] F. Fauberteau, S. Midonnet, and L. George, "Laxity-Based Restricted-Migration Scheduling," in *Proc. of the 16th IEEE ETFA*. IEEE Computer Society, 2011, pp. 1–8.

[6] G. Koren and D. Shasha, "D^{over}: An Optimal On-Line Scheduling Algorithm for Overloaded Uniprocessor Real-Time Systems," *SIAM J. Comput.*, vol. 24, no. 2, pp. 318–339, Apr. 1995.

[7] M. Chetto, D. Masson, and S. Midonnet, "Fixed priority Scheduling strategies for Ambient Energy-Harvesting embedded systems," in *Proc. of GreenCom*, 2011, pp. 50–55.

[8] M. Qamhieh, S. Midonnet, and L. George, "A Parallelizing Algorithm for Real-Time Tasks of Directed Acyclic Graphs Model," in *Proc. of WiP RTAS*, 2012.

[9] S. K. Kato, R. R. Rajkumar, and Y. Ishikawa, "A Loadable Real-Time Scheduler Suite for Multicore Platforms," Tech. Rep., 2009.

[10] M. Holenderski, M. v. d. Heuvel, R. Bril, and J. Lukkien, "Grasp: Tracing, visualizing and measuring the behavior of real-time systems," in *Proc. of WATERS*, 2010.

[11] M. G. Harbour, J. J. G. García, J. C. P. Gutiérrez, and J. M. D. Moyano, "MAST: Modeling and analysis suite for real time applications," in *Proc. of ECRTS*, 2001.

[12] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: a Flexible Real Time Scheduling Framework," in *Proc. of SIGAda*, 2004.

[13] R. Urunuela, A.-M. Déplanche, and Y. Trinquet, "STORM: a Simulation Tool for Real-time Multiprocessor Scheduling Evaluation," *GDR SOC SIP*, p. 1, 2009.

[14] P. Courbin and L. George, "FORTAS: Framework fOr Real-Time Analysis and Simulation," in *Proc. of WATERS*, 2011, pp. 21–26.

[15] R. Jayaseelan, T. Mitra, and X. Li, "Estimating the Worst-Case Energy Consumption of Embedded Software," in *Proc. of RTAS*, 2006, pp. 81–90.

[16] E. Bini and G. C. Buttazzo, "Measuring the Performance of Schedulability Tests," *Real-Time Syst.*, vol. 30, no. 1-2, pp. 129–154, May 2005.

Compositional Performance Analysis in Python with pyCPA

Jonas Diemer, Philip Axer, Rolf Ernst
Institute of Computer and Network Engineering
Technische Universität Braunschweig
38106 Braunschweig, Germany
{diemer|axer|ernst}@ida.ing.tu-bs.de

***Abstract*—The timing behavior of current and future embedded and distributed systems becomes increasingly complex. At the same time, many application fields such as safety-critical systems require a verification of worst-case timing behavior. Deriving sound guarantees is a complex task, which can be solved by Compositional Performance Analysis. This approach formally computes worst-case timing scenarios on each component of the system and derives end-to-end system timing from these local analyses. In this paper, we present pyCPA, an open-source implementation of the Compositional Performance Analysis approach. Targeted towards academia, pyCPA offers features such as support for the most common real-time schedulers, path analysis for communicating tasks, import and export functionality, and different visualizations. Thus, pyCPA is a valuable contribution to the research domain.**

I. Introduction

Embedded applications such as complex, distributed control-loops and safety-critical sensor-actuator interactions are subject to hard real-time constraints where it must be guaranteed that certain functions will finish before their deadline. In most cases, it is not straightforward to show that all timing requirements are satisfied under all circumstances. Research in the field of real-time performance analysis and worst-case execution time analysis provided various formal approaches such as compositional performance analysis (CPA) [1] to solve this problem. CPA breaks down the analysis complexity of large systems into separate local component analyses and provides a way to integrate local performance analysis techniques into a system-level analysis.

This papaer presents pyCPA[1], an easy-to-understand and easy-to-extend Python implementation of CPA. To our knowledge, pyCPA is the only free (as in speech) implementation of the CPA approach. pyCPA targets academic use-cases such as lectures in real-time education, research of further extensions of CPA, or simple reference benchmarks for novel analysis methodologies. To ease interaction with other toolkits, pyCPA offers support for integration through file-based I/O with other related tools such as SMFF [2], SymTA/S [1] and MPA [3].

The philosophy of pyCPA is to include only a baseline implementation of research relevant algorithms (e.g. analysis of fixed-priority schedulers) without puzzling or distracting add-ons to keep the package as simple and handy as possible. Thus, contrary to commercial solutions such as SymTA/S, pyCPA does not include any industrial scheduling protocols such as CAN, Flexray, OSEK, and others. Nevertheless, pyCPA is built in a modular fashion and can easily be extended to support such protocols, too. pyCPA is not aimed for maximum performance, and it is not overly fine-tuned to keep the implementation simple and comprehensible. Only obvious performance tweaks are included.

The remainder of the paper is organized as follows: In Section II, we give an overview of real-time analysis approaches and corresponding analysis tools. Then, in Section III we elaborate on the system model as used in CPA and how it is implemented in pyCPA. After the formal foundation of CPA is introduced in Section IV, we sketch the workflow of pyCPA by analyzing an exemplary architecture in Section V. The integration of pyCPA with other toolkits such as SMFF is presented in Section VI . Finally, we conclude the paper in Section VII.

[1]http://code.google.com/p/pycpa

II. Related Work

There are different approaches for formal analysis of worst-case timing behavior on system level. Exact approaches like Uppaal [7] use model checking techniques to derive the worst-case timing of a system. This can be very expensive in terms of run-time and memory for larger (realistic) systems. Holistic approaches such as [8] have similar issues. Compositional approaches like Real-Time-Calculus [6] and Compositional Performance Analysis (CPA) [1] solve this by decomposing the analysis of the system at component level. They use abstract event models to describe the interaction of components in the worst- and best-case. Event models describe the maximum and minimum events arrivals for specific time intervals rather than exact instances in time. This can lead to pessimism in the analysis, but avoids the state space explosion from which holistic approaches suffer.

TABLE I
Tools for Worst-Case Timing Analysis

Tool	Approach	Commercial	Free	Open-Source
MAST	[4]	no	yes	yes
Uppaal	[5]	yes	yes	no
MPA Toolbox	[6]	no	yes	yes
SymTA/S	[1]	yes	no	no
pyCPA	[1]	no	yes	yes

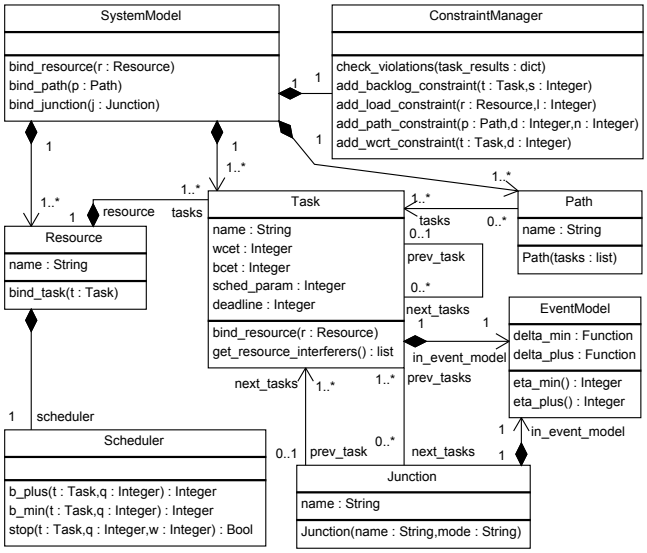


Fig. 1. System model of Compositional Performance Analysis

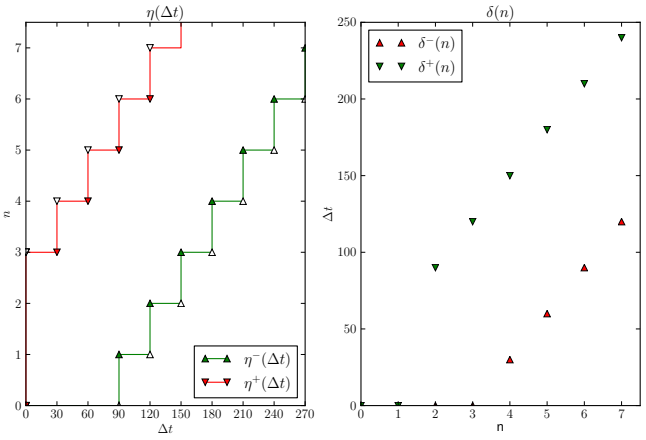


Fig. 2. Event model for a periodic activation with a period of $P = 30$ and a jitter of $J = 60$

Most of the proposed approaches have been implemented in software tools, which are summarized in Table I. With pyCPA, we present a toolkit which implements the CPA approach which is also used in the commercially available SymTA/S tool. pyCPA is publicly available in source-code, like the MPA Toolbox implementing Real-Time-Calculus.

III. The CPA System Model

In CPA, systems are modeled by sets of resources and tasks (see Figure 1). A resource provides processing time which is consumed by the tasks mapped to it. The mapping of tasks to resources is represented by references between tasks and a resource. Contention for resources with multiple tasks is resolved according to a scheduling policy (e.g. static priority preemptive), for which each task may include a scheduling parameter. The scheduling behavior is specified within a scheduler class which defines window functions (`b_min()` and `b_plus()`) used in scheduling analysis [9], see Section IV.

The execution behavior of a task τ_i is divided into the following steps: activation, core execution and finally completion/propagation. After being activated, a task (or job) is ready to execute and can be scheduled. It is assumed to require a core execution time in the interval between the best-case and worst-case execution times $[C_i^-, C_i^+]$. Between activation and completion, tasks may be interrupted by other tasks running on the same resource, which can be obtained via `get_resource_interferers()`.

A distributed application consisting of multiple communicating tasks is implicitly described by a directed graph (via the `next_tasks` and `prev_task` attributes) in which nodes are tasks and edges represent functional data dependencies. After a task's execution is completed, the task activates its dependent tasks (propagation). The application graph consists of task chains, which are called paths in pyCPA. Here, forks are possible, i.e. one task can activate multiple other tasks (forming multiple paths). The opposite, i.e. a join, is represented by a junction, and requires the definition of a semantic or "mode". There are two common join-semantics as discussed in [1]: For an *OR-join* any incoming event produces one outgoing event, and for an *AND-join*, an outgoing event is produced once events are available on all incoming edges.

Also part of the model are optional constraints, which can be used to define deadlines for tasks and path response times, limitations on the load of resources, or activation backlog (which usually translates to buffer requirements). In pyCPA all elements which model the system architecture (i.e tasks, resources, event models, junctions and paths) are distinct classes, as shown in Figure 1. For easy navigation, all classes are (redundantly) cross-referenced, e.g. resources keep a list of all mapped tasks and each task keeps a reference to its resource.

As discussed above, task activation and completion denote specific events, which are chained for dependent tasks (i.e. the completion event of one task is the activation event of its dependent tasks). Events can also originate from external sources, such as a timer. The arrivals of activation events of a task τ_i are modeled by minimum / maximum arrival curves $\eta_i^-(\Delta t)$ / $\eta_i^+(\Delta t)$, which return a lower / upper bound on the number of events that can arrive within any half-open time window $[t, t + \Delta t)$ [10]. These functions have pseudo-inverse counterparts, the so-called maximum / minimum distance functions $\delta_i^+(n)$ / $\delta_i^-(n)$, which return an upper / lower bound on the time interval between the first and the last event of any sequence of n event arrivals. Such event models cover all possible event arrivals of a specific event source as opposed to a specific trace of events.

For compact representation, standard event models in [10] use three parameters, event model period P, event model jitter J and a d^{min} which specifies the minimum distance between successive events in case the jitter is larger than the period.

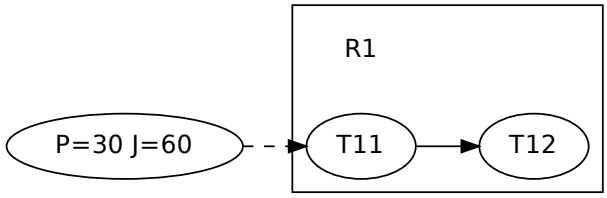


Fig. 3. A very simple pyCPA system model: two communicating tasks stimulated by one event model are mapped to one resource

The δ-functions for such an event model representation are as follows:

$$\begin{aligned} 0 \leq n < 2 \quad &: \quad \delta^+(n) = \delta^-(n) = 0 \\ n \geq 2 \quad &: \quad \delta^+(n) = (n-1)P + J \\ &\quad\;\; \delta^-(n) = \max((n-1)d^{min}, (n-1)P - J) \end{aligned} \tag{1}$$

Figure 2 shows the arrival curves and minimum distance functions for a periodic task activation with a period of $P = 30$, and a jitter of $J = 60$. In pyCPA, event models are internally described by their δ-functions, which are represented as actual function references. There are generator functions for typical event models, such as periodic with jitter or periodic bursts. The η-functions, which are needed during some analysis steps, are derived directly from the δ-functions by using the following transformation:

$$\begin{aligned} \Delta t = 0 \quad &: \quad \eta^+(\Delta t) = 0 \\ \Delta t > 0 \quad &: \quad \eta^+(\Delta t) = \max_{n \geq 1, n \in \mathbb{N}} \{n \mid \delta^-(n) < \Delta t\} \\ &\quad\;\; \eta^-(\Delta t) = \min_{n \geq 1, n \in \mathbb{N}} \{n \mid \delta^+(n+2) > \Delta t\} \end{aligned} \tag{2}$$

To speed up the event-model transformation, pyCPA leverages the fact that δ-functions are monotonous and implements a binary search. For further efficiency, δ-functions are cached, as they are referenced often with the same values (e.g. during event propagation, see next section). Note that although the CPA system model was conceived to analyze tasks executing on processor resources, it can also be used to for different systems such as Ethernet networks as presented in [11], [12] as well as CAN-buses as shown in [13]. Due to the CPA approach, pyCPA performs very fast, with the results being available within seconds. Even for a large system with over 1700 tasks on over 500 resources and an average load of over 90%, the analysis required only a couple of minutes.

A. Design Entry

Although pyCPA does not provide a GUI, there are rich ways to enter a system description. The easiest way is to instantiate the corresponding CPA objects directly in Python. As an example, consider the system model shown in Figure 3, which represents a small system with one resource R1 and two dependent tasks T11 and T12. Listing 1 demonstrates how this system can be represented in pyCPA. At first, a system object is instantiated which stores further objects. A resource

```
1  s = model.System()
2
3  r1 = s.bind_resource(model.Resource("R1",
4    schedulers.SPPScheduler()))
5
6  t11 = r1.bind_task(model.Task("T11", wcet=5, bcet=5,
7    scheduling_parameter=1))
8  t12 = r1.bind_task(model.Task("T12", wcet=9, bcet=1,
9    scheduling_parameter=2))
10
11 t11.link_dependent_task(t12)
12
13 t11.in_event_model = model.EventModel(P=30, J=60)
14
15 p1 = s.add_path("P1", [t11, t12])
16
17 s.constraints.add_backlog_constraint(t11, 5)
18 s.constraints.add_wcrt_constraint(t12, 90)
```

Listing 1. CPA system model directly instantiated in Python

named R1 is added to the system, for which a scheduling policy is defined by instantiating an SPP scheduler object. The scheduler object encapsulates scheduling specific functions, as discussed in Section IV. Both tasks are created and mapped to resource R1, worst- and best-case timing as well as the scheduling parameter (in this case a priority) are defined. Then, both tasks are linked according to the application graph and an input event model with a period of $P = 30$ and a jitter of $J = 60$ is specified for the first task. Since we are interested in the end-to-end latency from T11 to T12, we also define a path which includes the corresponding tasks. During runtime, pyCPA checks if the entered system description is well-formed e.g. there are no dangling tasks and no functional cycles without further external stimuli exist.

Some of the created tasks may exhibit constraints which either emerge from the underlying physical architecture (e.g. buffer size constraints) or non-functional timing constaints of the modeled application such as deadlines. In pyCPA, constraints are handled by a constraint manager (cf. Figure 1) which is attached to the system object. During analysis, the pyCPA kernel checks if any constraints are violated and eventually stops the analysis with an error message. As discussed, the constraint manager supports a set of constraints, but additional constraint semantics (e.g. reliability, mode-change latencies, slack) can be added by deriving from the pyCPA constraint manager class. In our simple example we constrain the available buffer size for the input queue of task T11 to 5 and add a deadline for task T12 of 90 time units.

Since one major focus of pyCPA is the interoperability with other timing toolkits, it offers a set of import and export libraries, which either convert a system description of another tool to the pyCPA model or vice versa. Specifically, pyCPA provides importers for SMFF [2] and SymTA/S 1.4 [1] as well as an exporter for MPA [3]. Depending on the complexity of the model transformation, additional import and export filters are quite simple to implement. Listing 2 illustrates the import of a SymTA/S 1.4 model.

```
loader = symload.SymtaLoader14()
s = loader.parse("symta14_test.xml")
```

Listing 2. Importing a SymTA/S 1.4 system model to pyCPA

IV. Compositional Performance Analysis

Once the system model is formulated, we are interested in its timing properties. A common metric is the worst-case response time, which is the largest time from activation of a task to its completion. Obviously, it is not straight forward to analyze timing for communicating tasks, since some event models are not know a priori (e.g. the input event model of $T12$ in Figure 3). Therefore, CPA uses an hierarchical iterative approach which is illustrated in Figure 4 to analyze the timing of such systems.

The general idea of the algorithm is as follows: At first, input event models for all tasks are initialized to one optimistic starting point. Naturally, the event model at the start of a path is an optimistic event model for all tasks on the path. Note, that later during the analysis, this optimism is resolved. Then, a local (component-level) analysis is performed for each task. After all local analyses have been performed, it is possible to derive output event models for all previously analyzed tasks. In a second step, newly derived output event models are propagated to all dependent tasks. If the output event model of a task has changed compared to the previous iteration step all tasks which are functionally or non-functionally (i.e. through resource sharing) influenced by this event model are re-analyzed.

This way, the two steps (local analysis and propagation) are alternated until either all event models remain stable or any constraint that might be specified (e.g. task deadline or path latency) is violated.

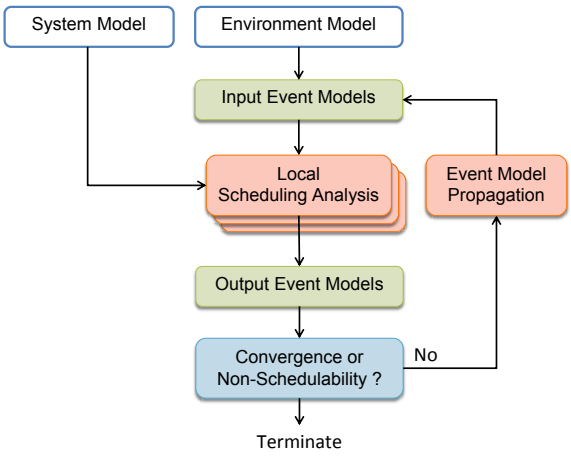


Fig. 4. The system analysis loop

```
class SPPScheduler(Scheduler):
    def b_plus(self, task, q):
        w = q * task.wcet
        while True:
            s = 0
            for ti in task.get_resource_interferers():
                if ti.scheduling_parameter <= \
                   task.scheduling_parameter:
                    s += ti.wcet * \
                    ti.in_event_model.eta_plus(w)
            w_new = q * task.wcet + s
            if w == w_new:
                return w
            w = w_new

    def stop(self, task, q, w):
        if task.in_event_model.delta_min(q + 1) >= w:
            return True
        return False
```

Listing 3. Simplified SPP scheduler class implementation

A. Local Analysis

The local analysis is based on a busy window approach as presented by Lehoczky in [14]. For this, we compute a so-called maximum q-event busy-time $B_i^+(q)$ which describes an upper bound of the amount of time a resource requires to service q activations of task τ_i, assuming that all q activations arrive "sufficiently early" (see [9]). A sufficient condition for the "sufficiently early" arrival of the q-th event is the arrival prior to the completion of its preceding event (the $(q-1)$-event busy-time). For this computation, a worst-case arrival of all interfering tasks is assumed. For a static-priority-preemptive (SPP) scheduler, the maximum busy-time can be computed as follows [14]:

$$B_i^+(q) = q \cdot C_i^+ + \sum_{j \in hp(i)} \eta_j^+(B_i^+(q)) \cdot C_j^+ \qquad (3)$$

where C_i^+ is the worst-case execution time of task τ_i, $hp(i)$ is the set of tasks with a higher-priority than task τ_i. Note that in this equation, $B_i^+(q)$ appears on both sides, resulting in an integer fixed-point problem. It can be solved by iteration, starting from $B_i^+(q) = q \cdot C_i^+$.

To find the worst-case response-time, only the first q_i^+ activations need to be considered, where q_i^+ is defined by a scheduler-dependent stopping condition. For SPP, the stopping condition is that all own and higher-priority load must be serviced. Hence:

$$q_i^+ = \min\{q \in \mathbb{N}^+ \mid \delta_i^-(q+1) \leq B_i^+(q)\} \qquad (4)$$

Note that in pyCPA, the satisfaction of the stopping condition is evaluated during the search for the worst-case response-time for every q without the explicit computation of q_i^+.

In pyCPA, this analysis is implemented in a modular way. As shown in the example from Listing 1, a scheduler-specific class object must be given for each resource. This class contains functions such as $B_i^+(q)$ and a stopping condition which evaluates whether the next activation has to be considered ($q+1 \leq q^+$) or whether the local analysis should terminate ($q+1 > q^+$). For an SPP scheduler a simplified

```
results = analysis.analyze_system(s)

for t in [t11, t12]:
  print("%s: wcrt=%d" % (t.name, results[t].wcrt))

bcl, wcl = path_analysis.end_to_end_latency(p1, 5)
print("Path latency: [%d,%d]"%(bcl,wcl)")
```

Listing 4. Analysis of a CPA system model

implementation is shown in Listing 3. Here, the busy-time function and the stopping condition are straightforward implementations of Equation 3 and Equation 4. pyCPA comes with scheduler implementations for the most common scheduling policies used in the embedded domain (e.g. static priority preemptive and non-preemptive, round-robin, TDMA, and earliest-deadline-first).

B. Global Analysis

The global analysis iteration is performed on task-level, i.e. the event-model propagation is done after the analysis of each task. For this, pyCPA maintains a set of dirty tasks to which all dependent tasks are added after the output event model of a task changes. In order to avoid unnecessary re-analysis of tasks, pyCPA analyzes tasks with the most dependent tasks first.

To compute the output event model of a task τ_i, we first need to determine its best- and worst-case response times R_i^- and R_i^+. These can be obtained from the busy windows which were gathered from the local analysis step. The worst-case response time can be found among the first q_i^+ busy-windows, whereas it is a safe assumption, that the best-case response-time equals the best-case execution time:

$$R_i^+ = \max_{q \in \mathbb{N}^+ \mid q \leq q_i^+} \left(B_i^+(q) - \delta_i(q) \right) \quad (5)$$

$$R_i^- = C_i^- \quad (6)$$

The worst-case scheduling jitter J_i^s for a task can be bounded to $J_i^s = R_i^+ - R_i^-$. From this, we can compute the output event model $\delta_{out,i}$ which adds the scheduling jitter to the input event model $\delta_{in,i}$ according to Equation 1.

$$\begin{aligned} \delta_{out,i}^-(n) &= \max\left(\delta_{in,i}^-(n) - J_i^s \ , \ (n-1)C_i^-\right) \\ \delta_{out,i}^+(n) &= \delta_{in,i}^+(n) + J_i^s \end{aligned} \quad (7)$$

This way of obtaining an output model is called *jitter propagation* in pyCPA. In [15], Schliecker et al. provide a more sophisticated event-model propagation which constructs the output event-model by considering the cases of all q^+ busy windows. This *busy-window propagation* yields tighter results, and therefore is the default in pyCPA.

V. Running an Analysis in pyCPA

Once a pyCPA system model is available, several different real-time metrics can be derived easily. Listing 4 shows the necessary steps to analyze the system model which was given in Listing 1 in Section III. The actual CPA iteration (local analysis and propagation) as depicted in Figure 4 is performed in `analyze_system()`. The analysis results are returned inside a result object and are available directly after the execution of `analyze_system()`. This includes the activation backlog, which is the largest amount of unprocessed task activation events, as well the best- and worst-case response-times. To derive more sophisticated properties of the system, such as the best- and worst-case end-to-end latency of a path, is is necessary to call dedicated analysis methods after the system has been analyzed. In Listing 4, we additionally derive the best- and worst-case path latency for 5 consecutive events for path P1.

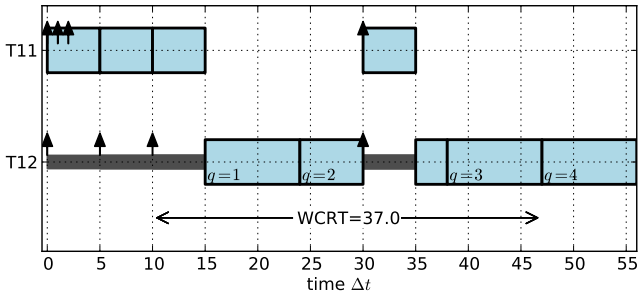


Fig. 5. Gantt-chart of the worst-case scheduling scenario for task T11

A. Visualization

Once analysis data is available, the results can be post-processed and visualized using one of the many existing Python packages such as matplotlib. Based on this, pyCPA provides several functions for visualization. The complete system model can be displayed using PyGraphviz. The system graph shown in Figure 3 was generated this way. The system plot is especially useful to visually inspect the entered system model for larger systems. Also, event models can be plotted using pyCPA (using matplotlib internally) as the one shown in Figure 2.

For illustrative purposes, it might be of further interest to generate one execution trace (Gantt-chart) which leads to the worst-case response time. For instance, the chart in Figure 5 shows the worst-case response time for task T12 from the initial example. pyCPA comes with a discrete event simulator which is built on top of SimPy. To generate the Gantt-chart, pyCPA simulates the critical instant behavior according to the specified scheduling policy and stores all preemption- and run-times of a task. The trace data can then be used to plot the Gantt-chart as shown in Figure 5 which also highlights the activation at which the worst-case response time is observed.

VI. Interoperability

pyCPA uses a generic system model that is compatible with that of many other tools. For instance, pyCPA provides an XML importer for SymTA/S system models. Furthermore, it can directly read and write system models generated by the system-models-for-free (SMFF) generator [2]. pyCPA uses

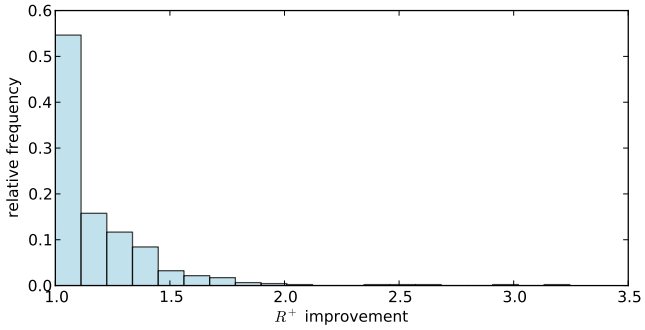


Fig. 6. Relative improvement of the busy-window propagation vs. jitter propagation

Python's `minidom` to parse the XML-based SMFF file-format and converts the SMFF model to the internal pyCPA representation. After analysis, the SMFF model can be written including annotated analysis results such as the worst-case response time as well as output event models.

To show the convenience and actual applicability of such an interface, we conduct an experiment in which the busy-window propagation from [15] is compared with the previously presented jitter propagation technique. Obviously, the results highly depend on the actual system model. Therefore, we use SMFF to generate a large set of representative random systems consisting of up to 4 resources and 3 paths with 4 tasks which lead to a resource load of up to 0.8. Actual mapping and scheduling parameters are randomized according to a heuristic implemented in SMFF. We have generated 500 systems with SMFF which were analyzed with pyCPA to derive the relative improvement of the busy-window propagation over jitter propagation. Systems, which were not feasible (i.e. WCRT larger than ten periods), were discarded. Figure 6 shows the distribution of this improvement. As expected, busy-window propgation yields a up to 3 times better result compared to jitter propagation. The analysis runtime for jitter and busy-window where approximately the same with about 10 ms per analyzed system. All experiments can be directly carried out in Python which has the immediate advantage that results can be post-processed and directly plotted. Thus experiments are self-contained and can be easily reproduced later.

It is also possible to export the pyCPA system model to other file-formats. This is useful to compare analysis results with other frameworks. For this purpose, pyCPA includes a simple exporter which directly outputs a Matlab description of the system to be used with the Modular Performance Analysis (MPA) framework. In our experiments, the analysis results for static-priority systems were identical with pyCPA and MPA, which matches our expectations.

VII. Conclusion

In this paper, we have presented pyCPA, a Python-based framework for Compositional Performance Analysis. It can be used to derive worst-case timing of complex embedded and distributed real-time systems. We have presented the basic architecture of pyCPA which is very easy-to-use as we have demonstrated in this paper by a small example. Furthermore, pyCPA is open-source and has a modular architecture, so it can be extended easily to cover different scheduling policies or implement advanced analysis algorithms. pyCPA also provides interfaces to existing tool suites such as the system-model generator SMFF or the Modular Performance Analysis framework. For these reasons, pyCPA is a valuable contribution to the research community in the field of timing analysis of embedded real-time systems.

Acknowledgment

This work has been funded by the "Bundesministerium für Bildung und Forschung" (BMBF), the "Deutsche Forschungsgemeinschaft" (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500 - spp1500.itec.kit.edu), the Advanced Research & Technology for Embedded Intelligence and Systems (ARTEMIS) within the project 'RECOMP', support code 01IS10001A, agreement no. 100202 as well as Intel Corporation.

References

[1] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System Level Performance Analysis–the SymTA/S Approach," *IEE Proceedings-Computers and Digital Techniques*, vol. 152, no. 2, 2005.

[2] M. Neukirchner, S. Stein, and R. Ernst, "SMFF: System Models for Free," in *2nd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, Porto, Portugal, July 2011.

[3] E. Wandeler, "Modular performance analysis and interface-based design for embedded real-time systems," Ph.D. dissertation, Swiss Federal Institute of Technology Zurich, 2006.

[4] M. Gonzalez Harbour, J. Gutierrez Garcia, J. Palencia Gutierrez, and J. Drake Moyano, "MAST: Modeling and analysis suite for real time applications," in *Real-Time Systems, 13th Euromicro Conference on, 2001.*, 2001, pp. 125 –134.

[5] R. Alur and D. Dill, "Automata for modeling real-time systems," *Automata, languages and programming*, pp. 322–335, 1990.

[6] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *ISCAS*, vol. 4, 2000.

[7] K. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1, pp. 134–152, 1997.

[8] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocessing and microprogramming*, vol. 40, no. 2-3, 1994.

[9] S. Schliecker, "Performance analysis of multiprocessor real-time systems with shared resources," Dissertation, Technische Universität Braunschweig, 2011, submitted 2010.

[10] K. Richter, "Compositional scheduling analysis using standard event models," Ph.D. dissertation, TU Braunschweig, 2005.

[11] J. Rox and R. Ernst, "Formal Timing Analysis of Full Duplex Switched Based Ethernet Network Architectures," in *SAE World Congress*, vol. System Level Architecture Design Tools and Methods (AE318). SAE International, Apr 2010.

[12] J. Diemer, J. Rox, and R. Ernst, "Modeling of Ethernet AVB Networks for Worst-Case Timing Analysis," in *MATHMOD*, Austria, 2012.

[13] P. Axer, M. Sebastian, and R. Ernst, "Probabilistic response time bound for can messages with arbitrary deadlines," in *Proc. of Design, Automation and Test in Europe*, Dresden, Germany, 2012.

[14] J. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *Proceedings of the 11th Real-Time Systems Symposium*, 1990.

[15] S. Schliecker, J. Rox, M. Ivers, and R. Ernst, "Providing Accurate Event Models for the Analysis of Heterogeneous Multiprocessor Systems," in *CODES-ISSS*, oct 2008.

Interoperable Tracing Tools

Luca Abeni, Nicola Manica
DISI - University of Trento, 38100 Trento, Italy
luca.abeni@unitn.it, nicola.manica@disi.unitn.it

***Abstract*—In order to provide reliable system support for real-time applications, it is often important to be able to collect statistics about the tasks temporal behaviours (in terms of execution times and inter-arrival times). Such statistics can, for example, be used for schedulability analysis, or to perform some kind of on-line adaptation of the scheduling parameters (adaptive scheduling, or feedback scheduling). This paper presents a set of software filters to extract such information from execution traces generated by different kinds of software. The presented software can be used to evaluate the real-time performance of a system or an application, to debug real-time applications, and/or to infer the temporal properties (for example, periodicity) of tasks running in the system.**

I. Introduction

Real-time systems are designed to respect some temporal constraints (generally expressed as deadlines) characterising the applications running in the system. This can be done in different ways, for example by providing a-priori schedulability guarantees (based on mathematical proofs that the temporal constraints will be respected), by analysing the system through simulations, by instrumenting a prototype of the system, or by performing some on-line dynamic adaptation of the scheduler behaviour. In general, all these approaches require some knowledge about the tasks behaviours or about important tasks parameters (execution times, inter-arrival times, dependencies between tasks, etc...), or require some kind of instrumentation or run-time monitoring of the tasks.

Such information are generally collected in the form of *execution traces*: sequences of various scheduling-relevant events associated to time, tasks, and CPUs. Execution traces can be generated by various tools (for example, by the operating system kernel, by a scheduling simulator, or by some user-space applications) and can be used for different purposes: for example, to check the correctness of a scheduling algorithm implementation, to infer the tasks behaviour, to collect statistics about the tasks, or to evaluate the system performance.

As an example of traces generated by the operating system kernel, the Linux kernel provides the *Ftrace* - function tracer - subsystem [1] that can generate configurable textual traces containing some relevant kernel events. This subsystem can be used to collect some statistics about the tasks execution, usable as an input for the schedulability analysis, or to infer information about the tasks temporal behaviour (for example, about task periodicity) [2]. The collected information can also be used for evaluating the performance of a real-time system (in terms of kernel latencies) or of a scheduling algorithm (in terms of tasks' response times), or to generate graphs describing the tasks' schedule. Tracing subsystems like Ftrace can be implemented in different operating system kernels (even μkernels such as Fiasco [3]), or in user-space applications such as the X server, to obtain information about how the clients' requests are scheduled [4].

Similar traces can be generated by scheduling simulators such as rtsim [5]. In this case, the traces are used to graphically visualise the scheduler's behaviour, or for evaluating the scheduler's performance. Statistics about the tasks parameters (eventually extracted from traces generated by an OS kernel) can be used as an input by the simulator.

Unfortunately, there is not a widely used tracing format, and different tools can generate different kinds of traces using different syntaxes (for example, Ftrace generates traces in text format, rtsim uses its own binary format, etc...). Even the same tool can change the tracing syntax in an non backward compatible way from release to release (for example, Ftrace recently changed the textual format of its generated traces, with the removal of the "`sched_switch`" tracer). Hence, some tool able to convert event traces between different formats is needed to increase the interoperability between the various tools. This paper presents TRCUTILS: a set of *trace filters* which allow to manipulate traces in different formats, converting them and extracting some useful information and statistics. For example, TRCUTILS allow to import Linux kernel traces in the rtsim visualiser, or to extract execution times and inter-arrival times probability distributions, to be used for stochastic analysis using some other tools.

The presented tools and methodology can be used either for collecting data to be analysed off-line (for example, for obtaining data to be used as input for an a-priori schedulability analysis), or for on-line adaptation (for example, for observing the scheduler behaviour in a feedback-scheduling system). These two use-cases will be described in this paper as *off-line mode* and *on-line mode*[1].

Off-line mode is generally used in 2 different ways:

1) the real-time tasks are run during the system design phase to profile them by dumping the scheduler's activities through the tracer, and extracting information about the execution and arrival times. Such information

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n° ICT-2011-288917 "DALi - Devices for Assisted Living".

[1]The on-line mode generally makes sense only for traces generated by an OS kernel.

are saved and used in a second time by some analysis tools that allow to properly design the system, using the correct scheduling algorithm and properly dimensioning the scheduling parameters [6].

2) the scheduler is simulated using a tool such as rtsim, and its activities are recorded in a trace. Such a trace can be post-processed to generate gannt diagrams of the tasks schedule, to compute tasks response times, or to evaluate some kind of system performance metric.

When working in on-line mode, the tracer and the tools which parse the traces are run in parallel with the real-time applications[2], and the generated data can be immediately used to identify real-time tasks and to adapt the scheduling parameters [7].

The rest of the paper is organised as follows: Section II briefly summarises the problems encountered when working with different tracing tools, Section III describes some related tools, Section IV introduces the TRCUTILS filters and the tracing pipeline, Section V presents some examples of TRCUTILS usage, and Section VI states the conclusions.

II. DIFFERENT KINDS OF EXECUTION TRACES

In general, a trace of the scheduler activities is a sequence $\mathcal{T} = \{e_i\}$ of *events* e_i (describing the tasks' execution) associated to various *attributes* describing the event type, the time when the event happened, the entities affected by the event, etc... These events are encoded in the trace according to a *syntax* which depends on the tool that generated the trace.

The differences between traces generated by different tools are not limited to the syntax only:

- each tracing tool produces different kinds of events described by different attributes, and hence containing amounts of information that depend on the specialisation of the tool. For example, a general-purpose OS kernel (such as Linux) can generate events about task creation/termination, context switches between tasks, task blocking/wakeup, and tasks migration, but has no information about job arrivals and job finishing. On the other hand, a scheduling simulator such as rtsim, being dedicated to handling real-time tasks, can generate events at job arrival times and job finishing times
- different tools can associate different amounts of information (in the form of attributes) to the same kind of event. For example, the "task wake up" events generated by the Linux kernel (through Ftrace) have a "CPU" attribute (indicating in which runqueue the task has been inserted when waking up), while rtsim does not associate any CPU to tasks wake ups
- some tools react to some scheduler actions by generating multiple events, whereas other tools generate one single event as a reaction to the same scheduler action. For example, in case of context switch between two tasks the Linux kernel (using Ftrace) generates a "context switch" event (characterised by the "prev task" and "next task" attributes), whereas rtsim generates two different "preemption" and "dispatch" events.

As a result,

- there is not a $1 \leftrightarrow 1$ relationship between the events contained in two traces generated by different tools
- when converting traces between different formats, information can be lost, or have to be guessed

TRCUTILS address these issues by using an intermediate *internal* format, which stores a maximum common denominator of all the information stored by different tracing formats. For example, the internal TRCUTILS format associates a "CPU" attribute to all of the events (so that information stored in Ftrace "wake up" events are not lost) and provides two different dispatch and preemption events (so that it is possible to model the amount of time spent by an OS kernel between preempting a task and scheduling the next one). When importing traces generated by Ftrace, a single "context switch" event generated at time t is converted in 2 events: "preemption" and "dispatch", both happening at the same time t.

The events stored in TRCUTILS traces are:

- *task creation*
- *task dispatching*
- *task preemption*
- *job arrival*
- *job termination*
- *deadline assignment*
- *deadline modification*

Tasks migrations are not associated to any specific event, because they can be inferred by looking at the "CPU" attribute of the dispatch events).

Each event is associated to the following attributes:

- The event type
- The task τ_i to which the event is referred (described by an integer number: the task identifier)
- The event time
- The CPU on which the event happens

The task creation events also store the task name, and the deadline-related events store the deadline values.

The TRCUTILS traces have no global header at the beginning of the trace, so that even fragments of traces can be correctly decoded and traces are suitable for streaming. Of course, incomplete traces can contain inconsistent sequences of events, but TRCUTILS can fix them: for example, if two consecutive "job arrival" events are received for the same task, it means that some events (such as the "job end" event) have been lost, and TRCUTILS introduce the missing events to make the trace consistent.

III. EXISTING TRACING TOOLS

There are various tools to trace and analyse the tasks execution, but none of them is particularly designed for interoperability between different tracing formats.

[2]Notice that the tracing tools can be scheduled at a priority lower than the priority of all the real-time tasks, to avoid interfering with their schedule.

RTDruid [8] is a development environment for ERIKA, composed by some Eclipse plugins. It allows the estimation of the worst case response time for different scheduling algorithms, and it includes an importer/exporter for various formats such as AUTOSAR XML. WindView [9] is specific for VxWorks and allows to collect and to visualize information about the execution of tasks. Feather-Trace [10] is a tool specific for the LITMUS project. It exports events to a trace files which can be analyzed by the unit-trace[3], which can perform a global-edf test, but no statistics are provided.

In [11] the authors developed a Linux kernel module able to store information about the current executing task in the system every scheduler tick. In this way is possible to measure and to understand the execution of the various tasks in the system. Few lines of the Linux kernel code must be modified to utilize the module.

Some TRCUTILS feature are currently under development to import and export data from and to some of the open-source tools mentioned above.

Other more generic tools are commercial, and are not a target for TRCUTILS interoperability. symTA/S[4] is a commercial scheduling analysis tool suite for different targets, like processors, buses, networks, ... The capabilities of the suite is tracing, statistical analysis of the trace and graphical report of the results. chronVAL[5] is a graphical tool which allows to analyze and validate embedded systems using the worst-case scenarios.

IV. THE TRACING PIPELINE

TRCUTILS provide a set of trace filters, organised in a pipeline as shown in Figure 1.

The first stage of the pipeline (the `import` filter) transforms the traces exported by external tools in the TRCUTILS internal format, which is used by the other stages of the pipeline. Currently, this filter is able to import traces generated by recent Linux kernel through Ftrace, traces in the old Ftrace format, rtsim traces, traces generated by an experimental version of the X server [4], and traces generated by the Fiasco μkernel through a patch provided by TRCUTILS. The `import` filter is able to process events as soon as they are generated, and can be used both in on-line and off-line mode.

The next stages of the TRCUTILS pipeline are composed by second set of filters that can:

- export traces in different formats (currently, only the rtsim format is supported as an export output), through the `export` filter;
- parse the internal format to gather statistics about execution times, inter-arrival times, response times, and system utilisation, through the `stats` filter;
- generate a chart (in `xfig` format) displaying the CPU scheduling;
- periodically display tasks statistics on the screen (like the "`top`" utility), through the `visual` filter;
- infer some of the tasks temporal properties, identifying (for example) periodic tasks.

When working in off-line mode, the filters read data from files and output the results to one or more files. In on-line mode, the filters communicate through standard Unix FIFOs (named pipes) and can be combined in different ways, to collect different kinds of information. For example, when working in on-line mode, the `visual` filter, which periodically displays important statistics for selected tasks (similarly to the standard "`top`" program) can be inserted in the pipeline. When working in off-line mode, the collected values are generally saved to files to be processed in a second time, but they can also be summarised by some statistics that are saved instead of the raw sequence of values, to save some disk space. Since connecting the different tools in a correctly working pipeline (creating all the needed FIFOs, etc...) can sometimes be difficult, some helper scripts have been developed.

The final stages of the TRCUTILS pipeline shown in Figure 1 generate some useful output from the internal trace (stored in TRCUTILS format) produced by the `import` filter.

The `export` filter can transform the trace in an `xfig` file displaying the schedule. This can be useful for visually analysing the scheduler's behaviour, and can be easily imported in various kinds of documents; for example, it has been previously used in various papers (for example, [4], [12]).

On the other hand, the `stats` filter can generate a table of values containing the *average*, *standard deviation*, *minimum*, and *maximum* values for standard metrics such as the *execution time*, *inter-arrival time*, *response time*, etc... The same filter can be used for generating various kinds of statistics, and the Probability Mass Function (PMF) of some observed values. Such PMFs are stored in an output format which is compatible with tools that allow to compute the probability to respect a deadline[6] [13].

Another filter, which periodically publishes some statistics through a Unix socket, is under development to be used as a server by programs performing dynamic adaptation of the QoS or of the scheduling parameters. Other filters which are currently under development allow the on-line identification of periodic processes.

Some of the filters are still under development, but a preliminary version of TRCUTILS has been released as open-source software (under the GPL) and is usable for researchers. Visit the TRCUTILS home page for more information and updates: http://www.disi.unitn.it/~abeni/TrcUtils.

V. EXAMPLES

This section shows some possible usages of TRCUTILS through some simple examples.

In the first example, the `export` filter is used to generate a diagram representing the tasks' schedule. If the input trace is produced by an OS kernel, the diagram can be used to analyse the scheduler's behaviour. In this example, Linux kernel traces (generated by Ftrace) are imported in

[3] http://cs.unc.edu/~mollison/unit-trace
[4] http://www.symtavision.com/symtas.html
[5] http://www.inchron.com
[6] See http://www.disi.unitn.it/~abeni/gamma-bound.tgz

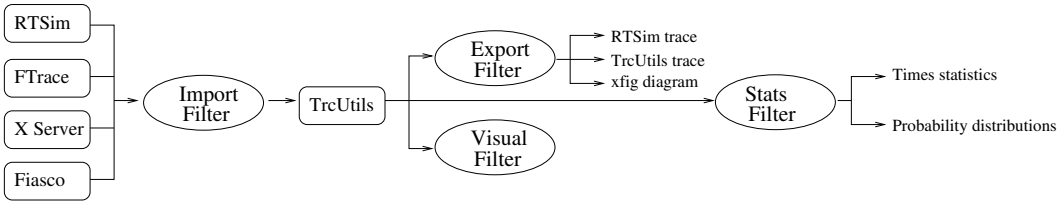


Figure 1. The TRCUTILS pipeline

the TRCUTILS format and exported to xfig or to rtsim (after eventually filtering the trace to select the relevant tasks and time interval) to generate the scheduling diagram. Three periodic tasks $\tau_1 = (30ms, 250ms)$, $\tau_2 = (20ms, 50ms)$, and $\tau_3 = (10ms, 100ms)$[7] have been scheduled on a dual processor system running the Linux kernel, capturing a scheduling trace through Ftrace and the scripts provided TRCUTILS. First, the trace has been converted in a TRCUTILS trace through the `import` filter and converted in a scheduling diagram (in `xfig` format) using the `export` filter. The resulting diagram is displayed in the top of Figure 2. Note that in this example the PID filtering feature of the `import` filter has been used to trace only the three periodic tasks (with PIDs `20461`, `20462`, and `20463`) and the `idle` task: all the events associated to tasks that are not selected by the PID filter have been associated to the `idle` task (also note that there is one idle task per CPU). The figure shows that all the tasks are started on CPU 0, and after a short time task τ_2 is migrated to CPU 1. After this, all the tasks are scheduled reasonably.

The same trace has been also exported to the rtsim format and displayed using the rtsim trace visualiser, as shown in the bottom of Figure 2. Since Ftrace produces times in μs and rtsim has problem to cope with such large numbers, the `export` filter converted times from μs to ms, dividing them by 1000.

In the second example, the `stats` filter has been used instead of the `export` filter, to collect some information and statistics for performance evaluation. The average, standard deviation, maximum value and minimum value of the inter-arrival, execution, and response times of the three tasks shown in Figure 2 have been computed, and are shown in Table I. From the table, it is possible to see that the kernel is pretty good at activating the tasks at the correct times (the average inter-arrival times match the expected values, and teir standard deviation is pretty small), but there is a large variation in the response times. Such a large variation is due to the fact that the three tasks have not been scheduled with real-time priorities. To better highlight this effect, the Cumulative Distribution Functions (CDFs) of the response times for the three tasks have been measured (by using a different option of the `stats` filter) and are displayed in Figure 3. All the results presented up to now can be obtained by just changing the latest stage of the processing pipeline.

Repeating the experiment with the proper real-time priorities (computed according to RM), the response times computed by the `stats` filter also matched the expected values.

[7] $\tau_i = (C_i, P_i)$ means that task i has worst case execution time C_i and period P_i

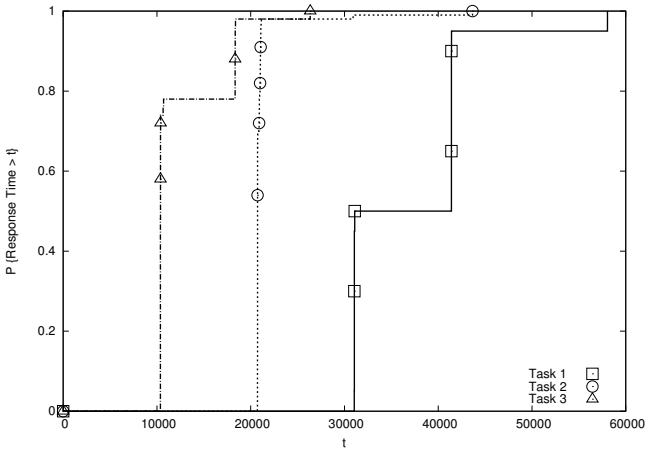


Figure 3. CDFs of the response times for 3 periodic tasks.

Table II
INTER-PACKET TIMES AS MEASURED IN THE SENDER. TIMES ARE IN μs.

Test	Average	Std Dev	Max	Min
T1	1190	29	1569	1040
T5	5198	22	5278	5058
T10	10195	22	10277	10062
T50	50207	27	50298	50081
T100	100207	25	100290	100093

A similar usage of TRCUTILS has been useful in the past to collect timing information about the IRQ threads in Preempt-RT [14]. This information have been used in previous works to properly dimension a CPU reservation to schedule such IRQ threads [15]. The third example shows this kind of TRCUTILS usage. First of all, the reliability of the information about IRQ threads gathered through TRCUTILS has been tested by sending a periodic stream of UDP packets between two computers, and measuring the inter-packet times in the sender (Table II) and in the receiver (Table IV). The tables report the results of 5 tests (`T1`, `T5`, `T10`, `T50`, and `T100`), where test `T`x indicates that the sender sends an UDP packet every x ms (note that the times in the tables are expressed in μs). Then, TRCUTILS have been used to extract the inter-arrival times of the network IRQ thread in the receiver machine, summarised in Table III. By comparing Table IV and Table III, it is possible to verify the correctness of the collected data.

After verifying the reliability of the measurements, some

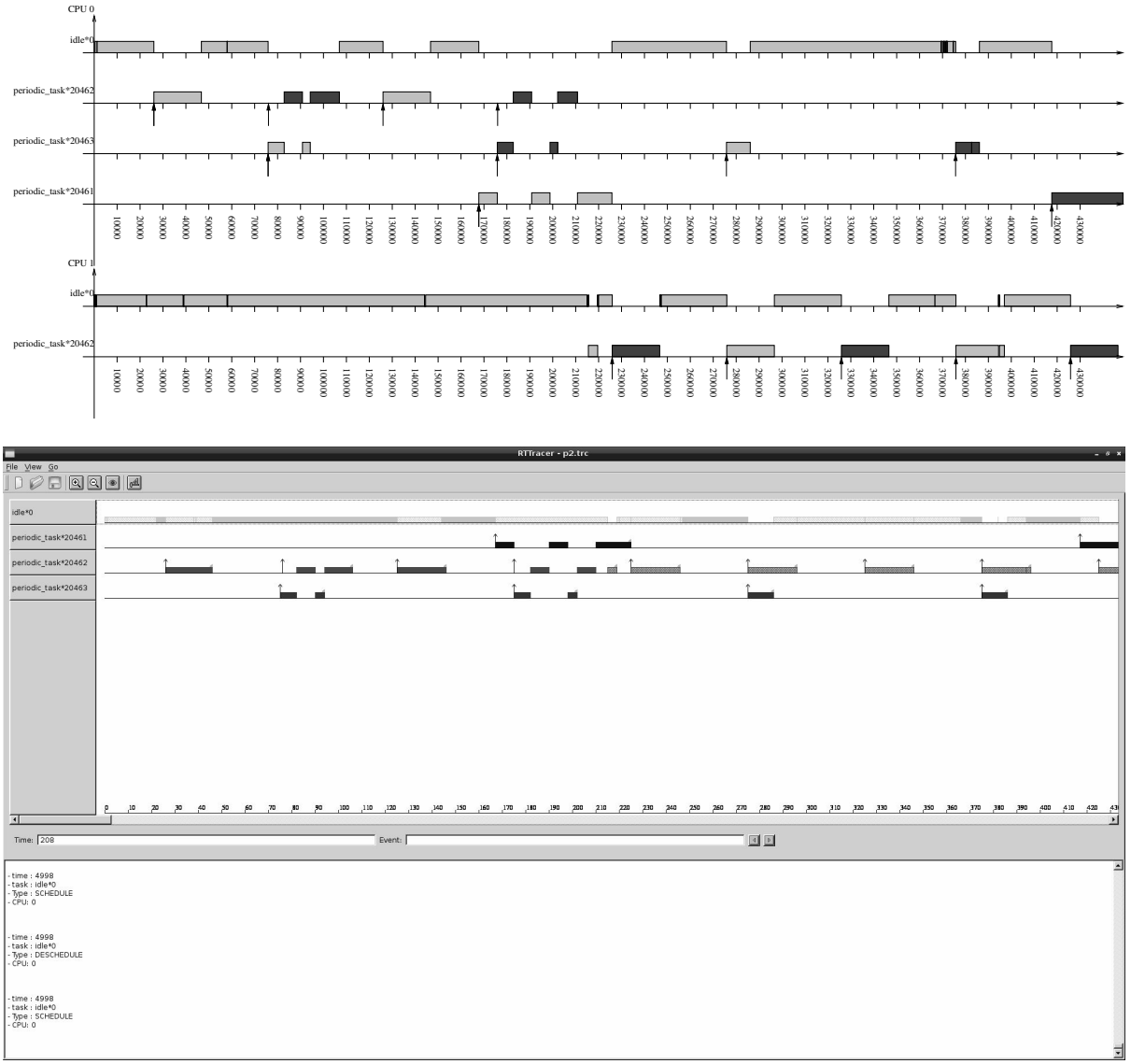


Figure 2. The Linux scheduler serving 3 periodic tasks, visualised by xfig and by the rtsim tracer.

Table III
INTER-ARRIVAL TIMES FOR THE NETWORK IRQ THREAD. TIMES ARE IN μs.

Test	Average	Std Dev	Max	Min
T1	1210	32	1424	59
T5	5222	117	5385	63
T10	10264	60	10353	10093
T50	50832	627	50353	50082
T100	100424	9342	100313	76

Table IV
INTER-PACKET TIMES AS MEASURED IN THE RECEIVER. TIMES ARE IN μs.

Test	Average	Std Dev	Max	Min
T1	1207	1011	14336	0
T5	5212	1019	6144	4096
T10	10210	271	12288	8192
T50	50229	1023	51200	49152
T100	100204	530	100352	98304

information about the network IRQ thread execution times (needed to perform some kind of performance analysis of the system) have been collected and are shown in Table V. Then, the `netperf`[8] tool has been used to produce a huge UDP traffic, and TRCUTILS have been used to measure the probability distributions of the execution inter-arrival times for the network IRQ thread. The Probability Mass Functions for such times are presented in Figures 4 and 5.

[8]http://www.netperf.org

VI. CONCLUSIONS

This paper describes TRCUTILS, a set of trace filters that can be used to convert execution traces between different

Table I
STATISTICS ABOUT THE THREE PERIODIC TASKS. TIMES ARE IN μs.

Task	Execution Time				Inter-Arrival Time				Response Time			
	Avg	Std Dev	Min	Max	Avg	Std Dev	Min	Max	Avg	Std Dev	Min	Max
Task 1	31053	4.910	31047	31060	249875	3.244	249870	249879	37065	6967.998	31051	58047
Task 2	20727	8.024	20687	20748	49974	1.351	49969	49981	21138	2480.460	20710	43661
Task 3	10367	10.233	10345	10380	99949	2.380	99947	99954	12303	3755.381	10374	26344

Table V
STATISTICS ABOUT THE EXECUTION TIMES OF THE IRQ THREAD. TIMES ARE IN μs.

Test	Average	Std Dev	Max	Min
T1	15	5	63	9
T5	19	1	68	18
T10	14	1	29	13
T50	16	2	28	15
T100	21	3	23	12

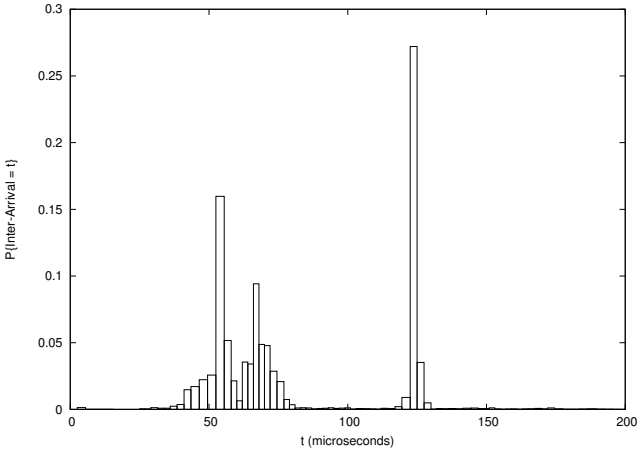


Figure 4. PMFs of the inter-arrival times for the network IRQ thread.

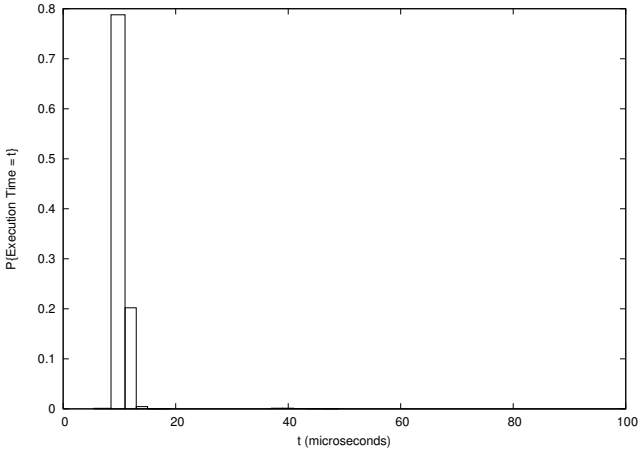


Figure 5. PMFs of the execution times for the network IRQ thread.

format, and to extract temporal information from such traces. The various filters are still under active development, and will be released soon under an open-source license.

While the software has not been officially released yet, it is reasonably stable and is able to successfully convert traces generated by various tools and to extract statistics from such traces.

More import and export filters are currently under development, and will be available in the first release of TRCUTILS.

REFERENCES

[1] S. Rostedt, "Finding origins of latencies using ftrace," in *Proceedings of the Eleventh Real-Time Linux Workshop*, Dresden, Germany, September 2009.

[2] P. Rallo, N. Manica, and L. Abeni, "Inferring temporal behaviours through kernel tracing," DISI - University of Trento, Tech. Rep., 2010.

[3] M. Hohmuth and H. Härtig, "Pragmatic nonblocking synchronization for real-time systems," in *USENIX Annual Technical Conference*, 2001.

[4] N. Manica, L. Abeni, and L. Palopoli, "Qos support in the x11 windows system," in *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, (RTAS 2008)*, St. Louis, MO, April 2008.

[5] L. Palopoli, G. Lipari, G. Lamastra, L. Abeni, G. Bolognini, and P. Ancilotti, "An object-oriented tool for simulating distributed real-time control systems," *Software: Practice and Experience*, vol. 32, no. 9, pp. 907–932, 2002.

[6] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," 1993, pp. 182–190.

[7] T. Cucinotta, L. Abeni, L. Palopoli, and F. Checconi, "The wizard of os: a heartbeat for legacy multimedia applications," in *Proceedings of the 7^{th} IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia 2009)*, Grenoble, France, October 2009.

[8] P. Gai, G. Lipari, M. Di Natale, N. Serreli, L. Palopoli, and A. Ferrari, "Adding timing analysis to functional design to predict implementation errors," 2007.

[9] D. Wilner, "Windview: a tool for understanding real-time embedded software through system vizualization," *ACM Sigplan Notices*, vol. 30, no. 11, pp. 117–123, 1995.

[10] M. Mollison, B. Brandenburg, and J. Anderson, "Towards unit testing real-time schedulers in litmusrt," in *Proceedings of the 5th Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2009, pp. 33–39.

[11] M. Asberg, T. Nolte, O. Perez, and S. Kato, "Execution time monitoring in linux," in *Emerging Technologies & Factory Automation, 2009. ETFA 2009. IEEE Conference on*. IEEE, 2009, pp. 1–4.

[12] L. Abeni, L. Palopoli, C. Scordino, and G. Lipari, "Resource reservations for general purpose applications," *IEEE Transactions on Industrial Informatics*, 2009.

[13] L. Abeni, N. Manica, and L. Palopoli, "Efficient and robust probabilistic guarantees for real-time tasks," *Journal of Systems and Software*, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121211003232

[14] S. Rostedt, "Internals of the rt patch," in *Proceedings of the Linux Symposium*, Ottawa, Canada, June 2007.

[15] N. Manica, L. Abeni, and L. Palopoli, "Reservation-based interrupt scheduling," in *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium, (RTAS 2010)*, Stockholm, Sweden, April 2010.

Advances in the automation of model driven software engineering for hard real-time systems with Ada and the UML Profile for MARTE

Julio L. Medina and Alejandro Pérez Ruiz

Departamento de Electrónica y Computadores, Universidad de Cantabria, 39005-Santander, SPAIN
{julio.medina, alejandro.perezruiz}@unican.es

Abstract

The traditional application of model based development techniques in the design of real-time systems comprises usually the generation of code from both structural and behavioral models. This work describes recent advances in a tool-aided methodology that enables the assembly and transformation of such design intended models into schedulability analysis models that match the corresponding automatically generated implementation code. Both, the analysis models and the code are generated by means of model transformations from the high-level architectural formalisms provided by the UML Profile for MARTE standard. As a novelty the Ada code generator uses not only the typical information provided in structural models, which is used to create the skeleton of the classes and procedures, but also its activities, whose scenario based behavioral information is used to fill the code inside the procedures and functions there contained. From the perspective of the real-time practitioner, the use of activities instead of state machines helps significantly to keep in tune the two fundamental views of the system: the implementation code, and its corresponding schedulability analysis model.

1. Introduction[1]

Model-based software development is progressively taking momentum in industry as one of the most promising software engineering approaches. It helps to create and keep assets of many kinds along the development process. It facilitates the separation of concerns, increasing the process efficiency, and finally empowering the quality of software.

For real-time applications, a model-based methodology can also help to simplify the process of building the temporal behavior analysis models. These models constitute the basis of the real-time design and the schedulability analysis validation processes. With that purpose, the designer must generate, in synchrony with the models used to generate the application's code, an additional parameterizable model, suitable for the timing validation of the system resulting out of the composition of its constituent parts. The analysis model for each part abstracts the timing behavior of all the actions it performs, and includes all the scheduling, synchronization and execution resources information that is necessary to predict the real-time qualities of the applications in which such part might be integrated. In the approach here presented, these analysis models are automatically derived from high level design models annotated with a minimum set of real-time features taken from the requirements of the application in which they are to be used. Following the generation of the application's code as a composition of the code of its constituent parts, the complete real-time analysis model of the application can also be automatically generated from the composition of the set of real-time sub-models that form it.

A discussion of such process used for the design of the real-time characteristics in a strict component-based development methodology may be found in [1].

The research effort that this paper presents considers the definition of schedulability analysis models as part of the chain of tools and techniques used in a model driven engineering approach. At this abstraction level, the concrete modeling paradigm used to conceive and elaborate the system is not specified, but for practical purposes we assume it is able to be expressed in UML[16]. This is a general purpose modeling language but we will use with it its standardized extensions for Modeling and Analysis of Real-Time and Embedded systems, namely the UML Profile for MARTE [13].

The very basic use of model based development techniques, not only in the design of real-time systems but in the software domain in general, comprises usually the generation of code from structural models like class diagrams. With those automations an initial set of skeletons of the classes and structural packages that form an application is usually easy to obtain. Also some form of reverse engineering is available through the usage of specially formatted "comments" placed as textual marks surrounding the space

1. This work has been funded by the European Union under contract FP7/NoE/214373 (ArtistDesign); and by the Spanish Government under grant TIN2011-28567-C03-02 (HI-PARTES). This work reflects only the author's views; the EU is not liable for any use that may be made of the information contained herein.

for the real code. The final implementation code is then inserted (usually typed by hand) between the marks managed by the code generators. A further refinement that generates both, specifications and bodies, in the modeling side, are code generators that use state machines for modeling the behavior of the classes. This mechanism uses the operations of a class as messages handlers that trigger the events between states. That way the messages from other objects can interact with the automaton of the class, though in a non-predictable order. Then, this kind of code generators is not consistent with the required wosrt case scenario-based description of activities used for schedulability analysis. For this reason a different approach to the code generation is necessary if we want to keep both models in tune.

The mechanism for code generation that may be used to fill the code inside the marks of the structural skeletons is the use of the behavioral models given for each operation of the class. These models are usually made for explanatory or documentation purposes, but they are well suited for specification. For this labor the proper modeling elements are activity diagrams. The formalization of the "code" inside actions may be either the standardized action language [17] of the OMG, or specific annotations in the target language with the actions to be performed.

This paper presents some advances in the methodology proposed and reports as a relevant contribution the definition and implementation of a new kind of code generator. It does not only generate the classical skeletons from UML classes and operations, but also fills the bodies of those operations with code generated from the interpretation of UML activities. The activities are graphically described using activity diagrams.

The paper is organized as follows: Section 2 presents a global view of the approach and situates the research efforts undertaken in its perspective. It also makes a brief summary of the challenges, and presents some related efforts as well as the basis of the modeling languages used for it. Section 3 summarizes the concrete rules for the code generation. It describes and identifies the intermediate formalisms in the modeling language for the generation of the implementation code and points out the technologies used for its automation. Section 4 presents a usage example that assesses the code generation features and illustrates the available results. Finally some conclusions and the definition of our next steps in the completion of the envisioned model driven engineering approach.

2. The approach

The approach that supports the efforts here described uses UML as modeling language. The UML standard extensions proposed by MARTE [13] for the modeling and analysis of real-time and embedded systems are used with it. It complements UML to enable the specification of the necessary real-time features in the models. A synthetic view of the approach is shown schematically in Figure 1.

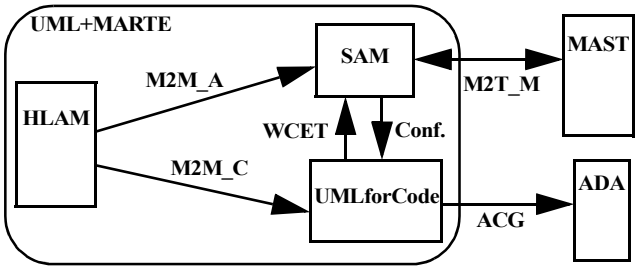


Figure 1. Models & transformations used in this approach

The initial model used to describe the application and its real-time features is constructed using the MARTE extensions for high level application modeling (HLAM). From this formalisms, two model-to-model (M2M) transformations are used. One, **M2M_A**, is used to create the UML representation of the analysis model. This transformation is used to create a model for each real-time situation under analysis together with the model of the processing resources, and the workload to consider. For this model the schedulability analysis modeling capabilities of MARTE (SAM) are used. The other, **M2M_C**, is used to generate an intermediate model useful for the code generation. In this methodology the target implementation language is Ada and the intermediate model, called **UMLforCode** in Figure 1, is a typical UML object oriented generic model that comprises structural as well as behavioral information. The behaviors of the operations in this model are expressed by means of activity diagrams.

The model-to-text transformation, denoted as **M2T_M** in Figure 1, is needed to generate the final schedulability analysis models in this approach, and it is part of our previous work [2]. An eclipse based tool [15] is provided for the generation of analysis models, the invocation of the analysis tools, and the retrieval of results back into the modeling analysis context. The tool then converts SAM models into the formalisms used by MAST [12] and then recovers its results back into the UML+MARTE model.

This paper presents the advances achieved in the techniques and tools used to generate the Ada implementation code from the UMLforCode object oriented generic model. This is a model-to-text transformation, called **ACG** (standing for Ada Code Generation) in Figure 1. The code implemented out of the combination of M2M_C and ACG is consistent from the execution semantics point of view with the analysis models generated out of the combination of M2M_A and M2T_M. Instrumented versions of the code will serve to meassure actual execution times (**WCET**) for

the SAM model. Once the analysis is performed, scheduling results are back annotated to the SAM models. These real-time configuration data include priorities (or relative deadlines) for the concurrent units, and priority ceilings (or preemption levels) for shared resources. Then, these data, denoted as **Conf.** in Figure 1, is used as part of the configuration information in the UMLforCode generation model.

2.1. The need for a new code generation technique

Following previous efforts that have studied the design of real-time systems using object oriented formalisms, we observe that most of them include the specification of the concurrency using structural models, usually at the design-for-implementation level. These dual structural-behavioral formalisms are made in the aim that this will help to realize schedulability analysis with the simple tasking model in mind and basic RMA techniques later on. Unfortunatly the complexity of the mechanisms used to generate the code makes this assumption not realistic, such as in ROOM [3] [4], Octopus/UML [8], ACCORD/UML [10] [11], Comet [7], or the design model extremely constrained and monolithic such as in HRT-HOOD [5], OO-HARTS [6].

Being a syncretism of all those mentioned, and in order to ease the application of simple schedulability analysis techniques, the high level application modeling constructs in MARTE (see its HLAM section in [13]) also facilitate the use of structural models for the specification of the concurrency. But the interactions between them (including distribution) may take complex patterns that require a richer model for the analysis. The offset based analysis techniques scale better to deal with this scenarios than the basic tasking model. HLAM proposes two basic building blocks, the real-time unit: *RtUnit* and the passive protected unit: *PpUnit*. As for the behaviors in them (the code inside the marks), due to its natural complexity it is usually not just passive linear code that can be modeled as a computation time; instead they include delays, and interactions among objects and nodes, mostly when they become formed out of a composition of distributed operations (behavioral models). In these cases a state machine is not directly transformable into an analysis model.

From the analysis perspective, the models that are required to apply the modern offset based analysis techniques, are fundamentally scenarios. A scenario is an expression of the (worst case) expected or observable manifestation of the design intents (coded behaviors). This is the basis for coping with complexity that distinguishes RMA schedulability analysis techniques from those other strategies based on timed automata or synchronous languages.

As a modeling language for this domain, the scheduling analysis modeling section of MARTE (SAM) is also able to express that kind of scenario models, and then it is an adequate formalism to feed the corresponding analysis tools. Unfortunately these scenarios are not necessarily part of the initial specification of the system behavior. They are a means to express: the expected stimuli, the high level expected workload, and the end-to-end timing requirements, but they are usually not the basic data used for design intent or code generation drawn by the designers.

The creation of these (usually worst case) analysis oriented scenarios in tune with the final code is actually the main duty and a high responsibility of the real-time practitioner. In order to help in this labor the automation tools need the model used for code generation to have the behaviors of its operations expressed as scenarios. For this reason the adequate input models for the generation of the code inside the operations in the UMLforCode model are UML activities. Then the tool that fills the code for the procedures and functions associated to the classes retrieves it from activity diagrams.

The use of scenarios has an additional benefit. This method helps to support the design of applications in terms of composable parts, which are closer in granularity to the concept of real-time objects than to the fully CBSE interpretation of components. In a fully component-based approach, the creation of the analysis models would have to be made as a combination of both, structural elements plus their deployment. In a model-driven approach, this later strong form of composability is in a higher level of abstraction, but still may benefit of the approach here described in order to assess a variety of non-functional properties, in our case of course the assesment of its timing properties by means of schedulability analysis.

3. The UMLforCode (meta)model

The purpose of having this intermediate model is basically to have a UML object oriented representation of the system that allows us to have the behaviors expressed in a way as close as possible to the way it is expressed its schedulability analysis model. Also this model must serve to implement the system in potentially different target programming languages. As a starting point for its practical implementation we have considered Ada as the target language.

In this section we describe the elements of UML that have been selected for the creation of these models, and the way they are used to generate Ada code. Instead of using a full metamodel, or a reduced version of the UML metamodel to formalize this description, we prefer to present it by identifying the capacities of the object oriented modeling/programming that are supported.

The technologies used for this automation are those provided by PapyrusUML as graphical tool, the UML2 plug-in

as model repository, and the Acceleo plug-in for the extraction of text from the UML2 models. As in marte2mast (M2T_M) [15], also here a number of Java functions have been necessary to implement the code generation.

3.1. Structural elements

The structural object oriented elements currently supported are Classes, Packages, and Interfaces. They are modeled in Class Diagrams.

- Packages may contain classes and have dependencies among them. Dependencies are implemented by means of **`with`** clauses between the ada **`package`** construct.
- Classes are the basic building blocks of code in an object oriented language. In Ada they are implemented by means of what Ada calls **`tagged types`**. These constructs support the inheritance and polymorphism, and hold in a natural way the UML concepts of object property (attribute) and operation (method). Static attributes and methods are declared out of the tagged type, so to keep them together they need to be hold by a wrapping Ada package in which the tagged type is also defined. This mechanism allows us to implement in Ada also the dependencies between classes and the visibility (accessibility) restrictions of the properties and operations. The inheritance and the realization of interfaces are implemented natively by Ada in the tagged types definition. Figure 2 shows how the Ada wrapping package visibility scopes match the visibility of the class members.

Figure 2. Class members visibility in the wrapping package

- Interfaces are directly implemented by using the corresponding Ada concept, which supports the definition of object methods. Static methods and attributes (including constants) are implemented like in classes by generating the corresponding code in the wrapping package. Interface object attributes are not supported.

Next we present some limitations of the tool, and modeling constraints for UMLforCode models. The tool is able to detect them and warn the user about their occurrence:

1. For members of a Class (properties and operations) the visibility clause `package` will not be enforced by the Ada language. They will be public and consistently renamed with the prefix **`package_`** .
2. Other visibility clauses (`public`, `protected`, and `private`) are supported as indicated in Figure 2.
3. Attributes need to have a name and a type.
4. Operations need to have a name and a type. Also each parameter needs: a name, the direction of assignment (**`in`**, **`out`**, or **`inout`**), and a type.
5. Classes and Interfaces need to have `public` visibility.
6. Nested classes are not supported.
7. Multiple inheritance is not supported in Ada. Interfaces realization is suggested to overcome this issue.

In order to handle inheritance, classes contained inside packages, and do so respecting the visibility defined by the modeler, three possible solutions where studied: (a) use the containing package directly as the wrapper, (b) use the class wrapping package as a child package of the container, (c) use the containing-contained relationship only as a mechanism to define the name of the wrapping packages for the class. The chosen solution was (c).

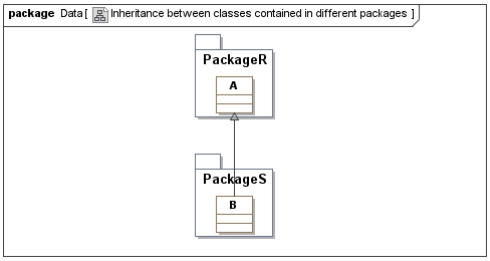


Figure 3. Inheritance among classes in different packages

To see this, consider the example in Figure 3. The wrapping package for class B will be denominated PackageS_B, correspondingly, the wrapping package for class A will be denominated PackageR_A. The fully qualified name of class B is PackageS_B.B and inherits from PackageR_A.A.

3.2. Behavioral models

Following the structure of SAM models described in previous research efforts [2] [14], MARTE provides concepts to organize the analysis models using three main categories: the platform resources, the elements describing the logical behavior of the system constituent parts, and finally the real-time situations (scenarios) to be analysed.

Scenarios are expressed usually by the annotation of SaSteps (SaCommStep, ResourceUsage or GaScenario) in sequence charts or activity diagrams. In marte2mast [15], scenarios may also be constructed from the lists of steps that are implicit in the chain of internal sub steps of a SaStep. These are expressed using the sub-usages list, hence using a structural element of the MARTE profile. This helps the tools to extract the analysis model in a more efficient way. But to express the high level end-to-end flows scenarios, sequence charts or activity diagrams are used instead.

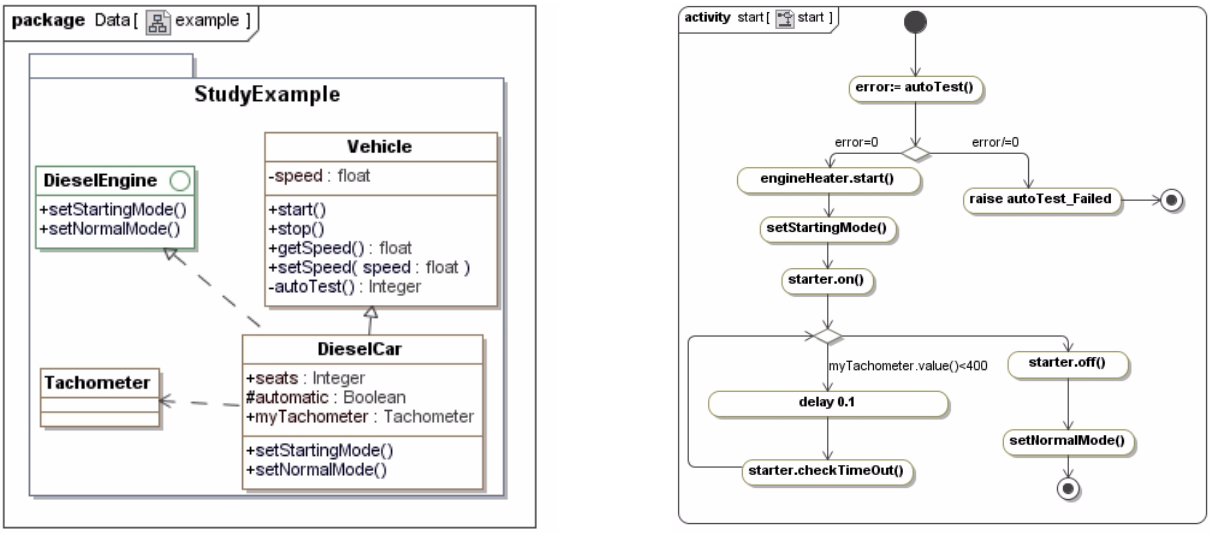


Figure 4. Structural and behavioral models used in the study example.

The elements that are currently used for the generation of the code inside class operations (bodies of the methods) are activities described by means of activity diagrams. The concrete modeling elements used in the diagrams are:

Initial nodes / Control Flow /Guards / Decision nodes / Opaque actions / Final nodes

These elements may be considered intuitively as corresponding to the basic assembly instructions for an Eckert-Mauchlyin architecture (also called Von Neuman architecture). With them the tool is able to extract out of the diagrams: regular **sentences**, **invocations,** simple **while loops** as well as **if-then-elseif-else** conditional branches (see the example in Figure 4). These basic constructs are the minimum required to describe scenarios, but they are sufficient for the general case; specially considering that, in the scope of the full approach, UMLforCode models are to be automatically generated by M2M_C from HLAM structural and behavioral models.

From its implementation point of view, in order to make code from activities, ACG has required much more than the basic automations provided by Acceleo. In particular due to the difficulties to handle variables inside the Acceleo scripts. The use of Java code inside the scripts, and the UML2 Java library created by the eclipse community, have been necessary to overcome this problem.

4. Study example

In order to show the capabilities of this tool we propose as an example the structural and behavioral models shown in Figure 4. They represent a very simplified extract of a car class and the activity model of an operation to start it.

Two pieces of code extracted from that model are shown next: the specification, and an extract of the implementation body of the `DieselCar` class. The implementation body shows the code for the function `start`.

The obtained Ada specification is:

```
with StudyExample_Vehicle.Vehicle;
with StudyExample_Tachometer.Tachometer;
package StudyExample_DieselCar is

    type Public_Part is abstract  new StudyExample_Vehicle.Vehicle
    and StudyExample_DieselEngine.DieselEngine  with  record
            seats : Integer;
            myTachometer : StudyExample_Tachometer.Tachometer;
    end record;

    -- Complete_type
    type DieselCar is new Public_Part with private;

    -- Public methods:
    procedure startIgnition (Self : DieselCar'Class );
    procedure setNormalMode (Self : DieselCar'Class);
    procedure setStartingMode (Self : DieselCar'Class);
    overriding procedure start (Self : DieselCar'Class);
    overriding procedure stop (Self : DieselCar'Class);

private
     -- Protected attributes
    type DieselCar is new Public_Part with record
            automatic : Boolean;
    end record;

end StudyExample_DieselCar;
```

The Ada body obtained for the start function inside DieselCar Class is:

```
package body  StudyExample_DieselCar is

    --Methods

    overriding procedure start (Self : DieselCar'Class) is
    begin
         error:=autoTest();
         if error/=0 then
            raise Autotest_Failed;
         elsif(error=0) then
            engineHeater.start();
            setStartingMode();
            starter.On();
            while myTachometer.value()<400 loop
                delay(0.1);
                starter.checkTimeOut();
            end loop;
            starter.off();
            setNormalMode();
         end if;
```

```
    end start;
    --Methods

    procedure setStartingMode (Self : DieselCar'Class) is
    begin
         --  (AP) Generated: replace with real body!
        pragma   Compile_Time_Warning   (True,   "setStartingMode
unimplemented");
        raise Program_Error;
        return setStartingMode (Self);
    end startIgnition;

    ...

end StudyExample_DieselCar;
```

5. Conclusions and future work

This work has presented the recent advances in a tool-aided methodology that enables the assembly and transformation of high level design intended UML models into schedulability analysis models that match the corresponding automatically generated implementation code. Both, the analysis models and the code are generated by means of model transformation from the high-level architectural formalisms provided by the UML Profile for MARTE standard. As a novelty this paper presents a new kind of Ada code generator that generates not only the skeleton of the classes, but also the code inside the procedures and functions there contained. It uses activity diagrams to fill them.

The necessity of this way of generating code lays in the fact that the UML+MARTE schedulability analysis specific models are described by means of scenarios. The creation of this (usually worst case) analysis oriented scenarios is actually the main duty of the real-time practitioner. Then, in order to automate the consistency between the code structure and the analysis model, both need to be expressed as scenarios, instead of state machines behaviors. From the real-time and embedded systems research community perspective, this effort constitutes another step to get the effective exploitation of the capabilities of the available analysis and verification techniques, which despite the efforts in dissemination, have not yet reached an audience large enough to reward the many years of work in the field. The modelling strategy and tools proposed in this work are another step in this direction.

References

[1] López P., Drake J.M., and Medina J.L., Enabling Model-Driven Schedulability Analysis in the Development of Distributed Component-Based Real-Time Applications. In Proceedings of 35th Euromicro Conference on Software Engineering and Advanced Applications, Component-based Software Engineering Track, Patras, Greece, August 2009, IEEE, ISBN 978-0-7695-3784-9, pp. 109-112.

[2] J. Medina and A. Garcia Cuesta. Model-Based Analysis and Design of Real-Time Distributed Systems with Ada and the UML Profile for MARTE. In Proc. of the 16th International Conference on Reliable Software Technologies-AdaEurope 2011, LNCS 6652, pp 89-102, ISSN 0302-9743

[3] Bran Selic, Garth Gullekson, and Paul T. Ward. Real-time Object Oriented Modeling. ISBN 0-471-59917-4, John Wiley & Sons, Inc., USA, 1994

[4] Bran Selic and Jim Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Rational white papers, http://www.rational.com/products/whitepapers/UML-rt.pdf, March 1998

[5] Alan Burns, Andy Wellings. HRT-HOOD, a structured design method for hard real-time ADA systems. ISBN 0 444 82164 3. Elsevier, Amsterdam, 1995

[6] Mazzini S., D'Alessandro M., Di Natale M., Domenici A., Lipari G. and Vardanega T. HRT-UML: taking HRT-HOOD into UML. In Proceedings of 8th Conference on Reliable Software Technologies Ada Europe, 2003

[7] Hassan Gomaa. Designing Concurrent, Distributed and Real-Time Aplications with UML. ISBN 0-201-65793-7, Addison-Wesley, USA, 2000

[8] E. Domiczi, R. Farfarakis and J. Ziegler. Octopus Supplement Volume 1. Nokia Research Center. http://www-nrc.nokia.com/octopus/supplement/index.html, 1999

[9] Laila Kabous. An Object Oriented Design Methodology for Hard Real Time Systems: The OOHARTS Approach. Doctoral Theses, School Carl von Ossietzky, Universität Oldenburg. 2002

[10] F. Terrier, G. Fouquier, D. Bras, L. Rioux, P. Vanuxeem and A. Lanusse. A Real Time Object Model. Presented in TOOLS Europe'96. Paris, France. Prentice Hall, 1996

[11] A. Lanusse, S. Gerard and F. Terrier. Real-Time Modeling with UML: The ACCORD Approach. In Selected papers from the First International Workshop on The Unified Modeling Language "UML"'98: Beyond the Notation. Mulhouse, France, June 3-4, 1998. Pp. 319-335. ISBN:3-540-66252-9. Springer-Verlag London, UK 1998.

[12] M. González Harbour, J.J. Gutiérrez, J.C.Palencia and J.M.Drake, MAST: Modeling and Analysis Suite for Real-Time Applications, in Proc. of the *Euromicro Conference on Real-Time Systems*, June 2001.

[13] Object Management Group, UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, version 1.1, OMG doc. formal/2011-06-02, 2011.

[14] J.L.Medina, M.González Harbour and J.M. Drake, Mast Real-Time: A Graphic UML Tool for Modeling Object-Oriented Real-Time Systems, in Proc of the *22nd IEEE Real-Time System Symposium (RTSS 2001)*, pp 245-256, 2001.

[15] http://mast.unican.es/umlmast/marte2mast

[16] Object Management Group. Unified Modeling Language version 2.4.1, OMG document formal/2011-08-06, 2011

[17] Object Management Group. Action Language for Foundational UML (Alf), Concrete Syntax for a UML Action Language. OMG document ptc/2010-10-05, 2010.

Enabling Model-Based Development of Distributed Embedded Systems on Open Source and Free Tools

M. Di Natale, M. Bambagini, M. Morelli
Scuola Superiore Sant'Anna, Italy

A. Passaro, D. Di Stefano, G. Arturi
Evidence Srl, Italy

Abstract

Model-Based Design brings the promise of an improved quality and productivity in the development of embedded systems and software. Flows based on commercial tools are today used in the industrial practice, albeit with several limitations. Furthermore, the analysis of the time properties considering scheduling and communication delays requires the addition of custom blocks to functional models, violating a desirable separation of concerns between the functional and the platform designs. Finally, to get full control at all stages in the modeling of systems and the automatic generation of implementations, it would be desirable that the toolchain is available as open source in all its components. We outline the initial steps in the development of a framework, largely based on the Scicos and Eclipse EMF open frameworks, that aims at providing support for the modeling, simulation, analysis and automatic generation of implementations of embedded functions on complex, distributed architectures.

1 Introduction

The use of models for the advance analysis of the system properties and verification by simulation, the documentation of the design decisions and possibly the automatic generation of the software implementation is an industrial reality, backed by several commercial products. The Model-Based Design approach (MBD) prescribes models based on a synchronous (reactive) execution model. Examples of available commercial tools are Simulink [5] and SCADE [7]. These tools are feature-rich and allow the modeling of continuous-, discrete-time, and also hybrid systems in which functionality is represented using an extended finite-state machine formalism.

However, MBD commercial tools lack in the capability of modeling physical computing architectures (and to some degree also task and resource architectures), as well as computation and communication delays that depend on the platform. Also, available commercial code generators typically provide implementations only for code to be deployed on a single CPU (exceptions are the Rubus Component Model and the accompanying tools [2]). AADL languages also supports execution platform modeling [3], but the tools originating from these languages seldom allow for simulation and automatic code generation (one example is [4]). To fill this gap, designers provide an implementation for a distributed execution platform by adding custom-developed communication code and performing the application partitioning by hand.

The analysis of computation and communication delays can be performed using the Truetime blockset in Simulink and the generation of platform-dependent code (including the task structure, the I/O and the communication code) can be obtained with custom blocks (for example, a well-known solution in the automotive domain is the RTI blockset from DSpace). However, both solutions create a model in which the functional solution is interspersed with platform-specific implementation blocks. If the implementation platform, or the task placement, or in general the task configuration is modified, the user must change all the affected blocks inside the model. Furthermore, code generation blocks for I/O and communication tend to be platform-specific.

Separation of the functional and platform models is advocated by many: examples from the academia are the Y-chart [15] and the Platform-based design [17] approaches. The OMG (a standardization organization) in its Model-Driven Architecture (MDA) [8] defines a three stage process in which a Platform-Independent Model or PIM is transformed in a Platform-Specific Model or PSM by means of a Platform Definition Model (PDM). Finally, the automotive industry AUTOSAR standard [1] defines a virtual integration environment for platform-independent software components and a separate model for the (distributed) execution architecture, later merged in a deployment stage (supported by tools). Unfortunately, the AUTOSAR metamodel is public but not open (use is only authorized to members of the consortium). Similarly, the Eclipse EMF-based Artop tool that provides the basic support for the AUTOSAR metamodel and its serialization is open but restricted for contributions and use to the consortium members. Most impor-

tant, AUTOSAR does not have any feature for modeling the behavior of the functions. Therefore, an exetrnal tool or the actual code is needed for functional modeling.

An open source or free framework can be constructed leveraging the following toolsets:

- Scicos [6] can be used for the functional modeling. The graphical language of Scicos is based on a synchronous semantics that is close to the one of Simulink. The toolset is open, offers a scripting language interface and a TCL-based graphical front-end. Unfortunately, Scicos currently does not support the modeling of Finite State Machine blocks. In addition, the code generation available for the dataflow part of Scicos has several limitation: it generates one function call for each block, does not provide support for all datatypes, and only generates single-task implementations for single rate models. In addition, the generated code is derived from the code used by the simulator with the overhead for all the hooks and data structures that are useful at simulation time but not needed on the target.
- The Platform model, as well as the model of the tasks and messages, can be generated using the Eclipse EMF modeling framework [11]. EMF offers the possibility of defining custom metamodels using its Ecore facility. Alternatively, several tools leveraged the EMF to build UML or SysML [9, 10] modeling environments. The most popular among them are Topcased and Papyrus. SysML modeling has the advantage of being conformant to a standard, with graphical editors and a number of additional tools for supporting the management of models. Unfortunately, SysML (and UML) are generic modeling languages, with little support for (embedded) domain-specific definitions. Therefore, a specialized profile is needed before it can be used for modeling embedded platforms and systems. Such a profile is currently available from the OMG, as MARTE (Modeling and Analysis of Real-Time and Embedded systems) [16]. However, MARTE is of difficult use, often cumbersome and lacking in the availability of concepts for modeling physical communication devices, as well as messages.
- A code implementation can be generated from EMF models (Ecore, or the UML/SysML models of tools that are based on Ecore) using model-to-text generators (of which model-to-code is a special case). Among them, the Acceleo toolset [12] is open, based on OMG standards, and allow to write code generator templates that query the model using statements in the standard OCL language.

The resulting methodology is a merger of the MBD and MDA paradigms. The overall tool flow is represented in Figure 1. The functional modeling, simulation and possibly verification of functional properties is performed using Simulink or Scicos. when simulink is used, the correcial code generators from Mathworks can be used to produce the behavioral code. Alyternatively, the functional model is exported in the Ecore XMl format and inported in the Eclipse EMF. Here, it can be extended with the platform and mapping models (in Ecore or using SysML) and the platform-dependent portions of the code produced. In the following sections we provide more details on the technologies and tools that have been developed for this purpose.

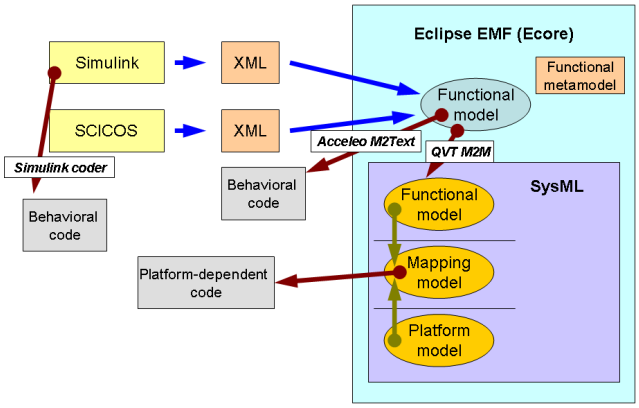


Figure 1. Exchange of information and code generation by the framework tools.

2 A Finite State Machine modeling tool for Scicos

To provide Scicos with the capability of modeling and simulating Finite State Machines, Evidence srl developed a modeling front-end, a simulation engine, a custom block which connects a Scicos model to the simulation engine, and a code genarator from a FSM specification.

The modeling tool (a snapshot is shown in Figure **??**) allows the specification of hierarchical (extended) Synchronous FSMs. Compared with commercial alternatives, or even UML/SysML State Diagrams, the model has several limitations. For example, no exchange of events among concurrent states, no join transitions, no inner or outer transitions and no access to internal variables from possibly concurrent states are allowed. This is done on purpose, to keep the FSM semantics concise and simple, thereby simplifying the code generation and possibly the verification of the behavior.

The tool produces an XML output describing the state machine structure and a stub for the creation of a custom Scicos block, representing the FMS in a larger model. An

executable reads the XML description and provides a simulation stub that can be connected to the Scicos simulator through the custom block to co-simulate the FSM in the context of the Scicos model.

2.1 Code generation from FSM blocks

A code generator produces an implementation of the FSM for execution on an embedded platform, consisting of an initialization function, and a step function to be executed at runtime (realizing the output update and the state update functions). The current generator produces an implementation based on a single step function. The code generator only accepts FSMs with periodic activation events and the step function that realizes the FSM behavior runs at the greatest common divisor of the activation events. However, in the future, we plan to extend the generator options to include a possibly more efficient partitioning of reactions in multiple functions to be scheduled at different periods [21].

3 Importing the functional model in the Eclipse EMF

The core of the modeling framework is implemented in the Eclipse EMF. Here, the functional model is matched with a platform model. A task, message and resource intermediate model is created in the process.

The functional model is created by importing in EMF the Scicos model (Alternatively, an input from Simulink is also possible). An Ecore metamodel has been defined for the import process (Figure 2)

This metamodel is not too dissimilar from the one proposed in the GeneAuto project [20] (actually, a simplified version of it). Contrary to GeneAuto, our metamodel is immediately available as an Ecore definition and it is only the starting point for a code generation process. GeneAuto has the objective of building an open-certified code generator, but does not handle distributed systems (and of course, neither the separation of functionality and task model) and does not generate platform-specific code.

In order to provide a code generator of industrial quality (with signal typing, code inlining and port variable folding) the functional model is then processed by a set of Acceleo code generation templates that produce the code that is functionally equivalent to the model.

At this stage, the generated code is strictly functional. Two generation modes are offered. In the first, a single-task implementation for the entire system is produced. The second mode allows for a multitask code implementation. When this mode is selected, we require that the designer partitions its model (at some level in the design hierarchy) in a set of superblocks (equivalent to Simulink subsystems)

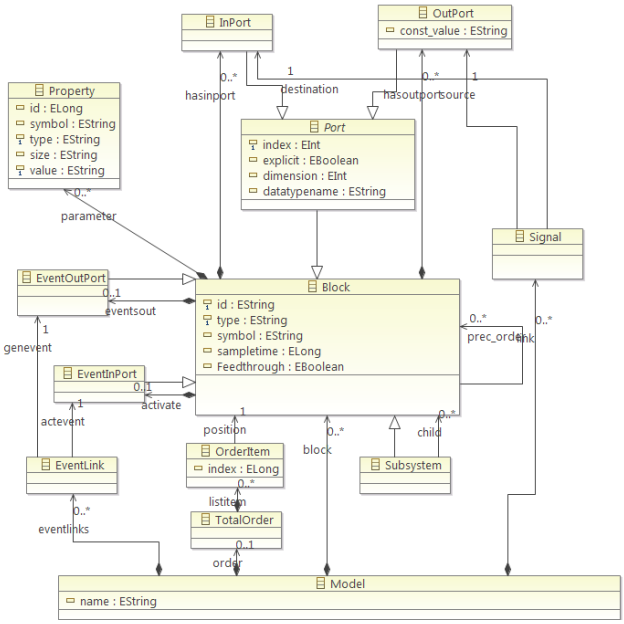


Figure 2. The Ecore metamodel for the functional part.

for which the execution is controlled by a single discrete-time clock. The superblocks are the units of execution for the code generation process. Each of the superblocks translates into a C function. Inside, the code implementing the blocks behavior (output update and state update) is generated and inlined, serialized according to the execution order generated by the Scicos modeler.

At runtime, access to the superblock ports does not happen in the way that is conventionally used by code generators. For example, the Simulink code generator project allows for several options, of which the most popular are to include the port variables in the signature of the Step function, or to use a set of global variables for the interface ports. Our generated code accesses the ports using a middleware-level API. Input or output ports are accessed using a simple and generic interface

```
EMW_read(sblock_id, port_id, port_type *read_var)
EMW_write(sblock_id, port_id, port_type write_expr)
```

This API eases the generation of the communication among superblocks in the case when these superblocks are mapped into different tasks or even remote nodes.

4 Matching the functional model with the platform model

Given that the functional model is produced by importing a description from Scicos, there is the need of a suitable

model (and metamodel) for the representation of the execution platform and the task and messages models. There are several options for the selection of such metamodels. In the following, we describe our current implementation, based on a set of custom metamodels, and then we outline another option, possibly more appealing for future work, in which the platforms and mapping models are generated by a customization of SysML.

4.1 Platform and mapping models based on custom Ecore metamodels

In our current implementation, the execution platform meta-model, shown in Figure 3, defines the hardware and software resources available in the system. The execution platform model needs to be modular and to support the concept of component libraries.

The main architectural elements that are used by the execution architecture designer are the following:

- *Embedded Control Unit* (ECU), which is a set of electronic boards, connected through communication links;
- *Board*, which may host several Controllers and Devices;
- *Controller*, which may include several Cores and Peripherals;
- *Core*, representing a computational unit;
- *Peripheral*, representing an electronic component which extends the Controller's functionalities;
- *Devices*, which represent I/O devices using a set of peripherals. So far, the meta-model supports buttons, touch screens, leds, lcd displays and servo motors. Communication units belong to a special class of devices, handled separately;
- *Real-Time Operating System* (RTOS), which may run on Cores.

The platform meta-model is completed by the project-specific definitions, showing the elements of the hardware and software architecture deployed for supporting the execution of the functions in one specific project instance. The main entities at this level are ECU instances (*ECUDeployment*), with the RTOSs executing on them (*RTOSDeployment*) and the communication buses (*Bus*). Each Bus object references the connected ECUs and the associated communication devices.

Figure 5 shows a screenshot of the developed Eclipse plug-in which defines the platform execution model out of the library components.

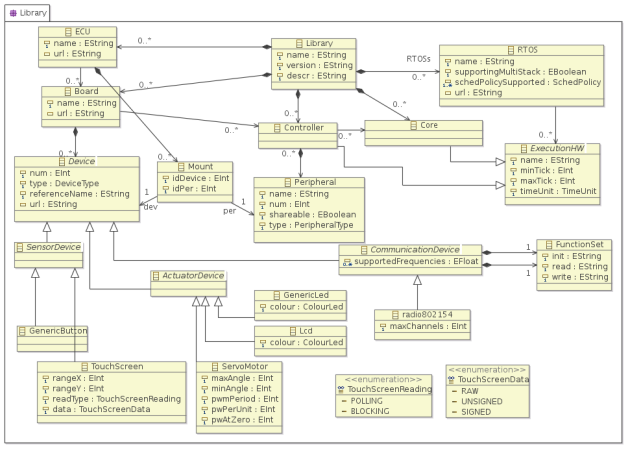


Figure 3. The meta-model for the library components that are used to construct the execution platform.

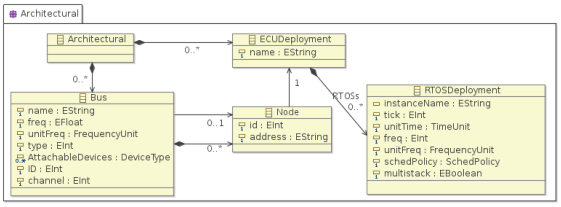


Figure 4. Meta-model of the objects that are used to build the project-specific execution architecture.

Once the system execution architecture is defined, a mapping model associates functional elements to tasks and then tasks to the (HW) processing elements, following the schema of Figure 6. The communication signals of the functional view are mapped (when needed) to messages and in turn, messages onto the physical links of the execution platform.

The task is the unit of concurrent execution that can run on one of the system cores, under the control of an operating system. The information about which RTOS hosts a specific task is handled by an association between the Task objects (Figure 6) and the RTOSDeployment (Figure 4). The Step methods of the functional subsystems are executed in the context of one of the task defined in the mapping model. More precisely, a list of Superblocks (defined as ProcMaps in the block-to-task mapping), sorted according to an execution order, belongs to each task. Each mapped Superblock refers to the appointed Step method. Moreover, designers must specify all the information concerning task times (such as period and deadline) and scheduling (priority and

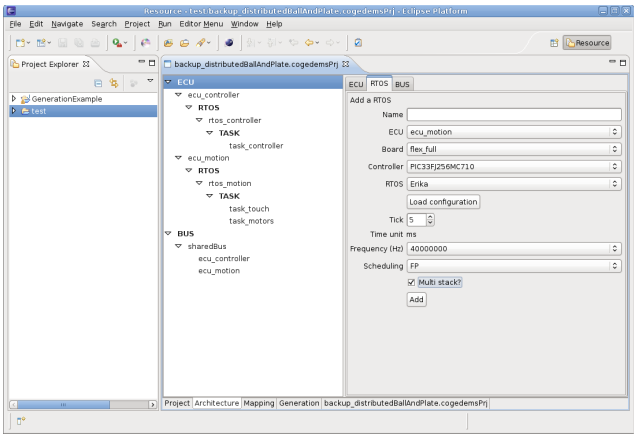


Figure 5. The editor used to construct the model of the execution platform.

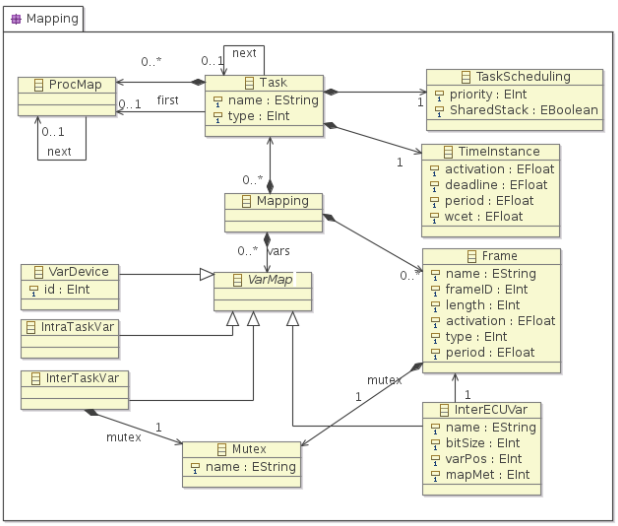


Figure 6. Meta-model for the definition of the mapping of functional components into tasks and connections among them.

reserved stack). The mapping of the functional subsystems (Procs) into tasks is subject to constraints and validation.

VarMap, the second important class into the Mapping package, concerns the mapping of signal variables or Vars (representing communication or I/O links, depending on the mapping). As shown in Figure 6, VarMap is a generic interface and four possible derived classes are instantiable: VarDevice is used to map custom devices to real devices and the other three to model all the possible communication scenarios according to the placement of the methods of Procs into tasks. *IntraTaskVar* communication takes place when two communicating procs are mapped into the same task (implemented using variables local to the task). *InterTaskVar* communication occurs between two tasks on the same Core. In this case, a suitable protection mechanisms for shared resources, provided by the operating system (part of the platform model) is used. The most complex case happens when the communication needs to be established between two remote tasks executing on different cores, or processors, connected by either an intra-chip network, a field bus or another network. An *InterECUVar* mapping object is automatically generated to represent this kind of communication, linking a Var to a specified portion of a *Frame*. Frames are periodic communication messages which are exchanged between two nodes through a shared bus.

4.2 Platform and mapping models based on SysML profiles and MARTE

A possibly more appealing option for the definition of the platform model and the mapping between functional subsystems and communication links and platform cores and communication or I/O interfaces consists in the use of a customized extension of the SysML metamodel. This allows us to use modeling tools with graphical editors, model checkers and processors, and possibly other standard tools. In this case, a stereotype of the standard SysML block concept could be used to represent Synchronous reactive subsystems and the standard SysML block definition diagrams (or bdd) and internal block diagrams (or ibd) could represent the block dependencies and the topology of communication. These models and diagrams, equivalent to the source Scicos or Simulink models, can be generated automatically starting from the Ecore model of the system functions, using a QVT transformation [19].

Next, the platform model could be constructed leveraging another set of SysML diagrams, using the MARTE stereotypes for computing nodes (cores), operating systems and computing resources, and suitable extending it for the model of the communication networks and protocols.

Finally, the mapping relationship can use the allocation tables of SysML, possibly complemented by an additional set of stereotypes to define the mapping of functional subsystems (their step functions) onto tasks and of signal data onto messages. Once again, MARTE provides standard stereotype concepts for representing tasks, shared resources and the attributes that are required by the most popular real-time analysis techniques.

5 Code generation and Example

The code generation of the platform-dependent code, including the tasking model and the communication code can be performed by processing the platform and mapping mod-

els using Acceleo (model-to-text) transformation templates. Acceleo is an open tool and allows great flexibility in the generation of the program implementation. In our current implementation, we are assuming an OSEK interface for the generation of the task model and the operating system-level API for task management and an AUTOSAR interface for I/O management. The tasks code is generated as a sequence of calls to the step methods of the functional subsystems mapped into it. Calls are sequentialized according to the mapping order, which must be consistent with the partial order of execution specified by the semantics of the functional model. Each invocation of the *step* method is preceded/followed by the respective middleware read/write functions implementing the functional data flows.

The code implementation of the middleware read/write functions, representing the communication links (*Vars*), depends on the task and core mapping of the blocks at the two endpoints of the communication.

For each communication signal, the Acceleo scripts can generate four different implementations. When the Var mapping is an instance of *IntraTaskVar*, the access functions execute read/write operations to a shared variable. If the endpoints are located on different tasks managed by the same operating system (*InterTaskVar*), read and write accesses to the shared variable are realized using a lock-free access protocol. The third case, which occurs when the sender and the receiver are on different processors (*InterECUVar*), is handled by using a middleware implementation wrapping a network communication API. Finally, communications between controller subsystems and the plant model can be mapped into I/O primitives.

A simple test case for the proposed methodology has been implemented by realizing a ball-and-plate controller with the same functional model (a PID controller) and two distributed implementations: one as a single node, and the other as a distributed platform (two nodes connected by a radio link). The two implementations have been obtained by changing the platform instance and the mapping specification and without writing a single line of code.

6 Conclusion

In this paper, we presented the overall architecture for a set of tools supporting a model-based development process for complex-distributed systems, from the system-level modeling of functions to the generated code

References

[1] *The AUTOSAR Standard, specification version 4.0*, the AUTOSAR consortium, web page: http://www.autosar.org.

[2] K. Hänninen, J.M. Turja, M. Nolin, M. Lindberg, J. Lundbäck, K.L. Lundbäck, The Rubus Component Model for Resource Constrained Real-Time Systems, 3rd IEEE International Symposium on Industrial Embedded Systems, Montpellier, France

[3] G. Raghav, S. Gopalswamy, K. Radhakrishnan, J. Hugues and J. Delange, Model based code generation for distributed embedded systems, European Congress on Embedded Real-Time Software, 19-21 May 2010, Toulouse, France.

[4] Hugues J., Zalila B., and Pautet L. Rapid prototyping of distributed real-time embedded systems using the aadl and ocarina. In 18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP'07), Brazil, 2007.

[5] "The Simulink product web page," http://www.mathworks.it/products/simulink/index.html.

[6] "The Scicos project web page," http://www.scicos.org/.

[7] "Scade product web page," http://www.esterel-technologies.com/products/scade-suite/.

[8] "The OMG Model Driven Architecture initiative," web page, http://www.omg.org/mda/.

[9] "The Unified Modeling Language (UML)," http://www.uml.org/.

[10] "The OMG Systems Modeling Language (SysML)," http://www.omgsysml.org/.

[11] "Eclipse Modeling Framework (emf)," http://www.eclipse.org/modeling/emf/.

[12] "The Acceleo model-to-text language and tool web page," http://www.acceleo.org/.

[13] "Real-time workshop/embedded coder," http://www.mathworks.com/products/matlab-coder/index.html.

[14] Y. Vanderperren and W. Dehaene, "From uml/sysml to matlab/simulink: current state and future perspectives," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, ser. DATE '06.

[15] B. Kienhuis, E. F. Deprettere, P. v. d. Wolf, and K. A. Vissers, "A methodology to design programmable embedded systems - the y-chart approach," in *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS*, 2002.

[16] "The OMG Marte Profile web page," http://www.omgmarte.org/.

[17] F. Balarin, L. Lavagno, C. Passerone, and Y. Watanabe, "Processes, interfaces and platforms. embedded software modeling in metropolis," in *Proceedings of the Second International Conference on Embedded Software*, ser. EMSOFT '02, 2002.

[18] "The eclipse modeling project," http://www.eclipse.org/modeling/.

[19] "The QVT transformation language," http://www.eclipse.org/qvt/.

[20] "The Gene-Auto project web site," http://geneauto.gforge.enseeiht.fr/.

[21] M. Di Natale, H. Zeng"Task Implementation of Synchronous Finite State Machines," DATE conference 2012.

SystemC based Simulator for Virtual Prototyping of Large Scale Distributed Embedded Control Systems

Alberto Ferrari, Marco Carloni, Alessandro Mignogna
Francesco Menichelli
ALES, Rome, Italy
{name.surname}@ales.eu.com

David Ginsberg
United Technologies Research Center
East Hartford, CT, USA
david.ginsberg@utrc.utc.com

Eelco Scholte
Hamilton Sundstrand
Windsor Locks, CT, USA
eelco.scholte@hs.utc.com

Dang Nguyen
Otis Elevator Company
Farmington, CT, USA
dang.nguyen@otis.com

***Abstract*—In this paper we present DESYRE, a SystemC-based virtual prototyping framework, that we have developed to build the simulator of a modern elevator system designed by Otis Elevator Company for large scale buildings. DESYRE aims at the simulation of industrial large-scale real-time distributed embedded systems and allows: the verification of the system functionality, taking into account the distribution over the network, the prediction of the performance, such as end-to-end latency, and the usage of the communication and computation resources for the entire range of scalability of the system. In the paper, we describe the framework details and its application to the construction of the virtual prototype of a scalable elevator system based on the CAN communication protocol. We show the tuning and validation of the simulated model against a test system composed of 24 physical nodes, linked to network and logic analyzers. We finally present results on the simulation of significant case studies (up to 20 floors and 8 cars, that corresponds to hundreds of interconnected nodes, having about 10 subcomponents each) to predict the scalability performances of the shared communication resources.**

Keywords - Distributed Systems, SystemC, Platform Based Design, Real-Time Systems, Virtual Prototyping

I. INTRODUCTION

Large distributed embedded control systems are becoming common even in consolidated and mature applications such as elevator systems. Elevator systems in very large buildings can contain tens of elevator cars servicing more than one hundred floors and can consist of several thousand networked electronic control units (ECU), each containing microcontrollers, input/output devices, actuators, network interfaces, and communicating over several shared communication busses [1]. Elevator system architectures sometimes need to be updated in order to realize the benefits which can be achieved through the incorporation of new technologies. However, it is likely that scaling limitations exist in any system architecture, and as a result there is a risk associated with committing to any particular new architecture without fully understanding these boundaries. To avoid expensive and time consuming modifications, it is crucial that the elevator system designer is confident that the elevator system can achieve correct operation prior to installation. In this work we describe a set of virtual prototyping tools which provide that ability, and allow the system designer to quickly perform cost optimization through evaluation of architectural tradeoffs while ensuring that all requirements are met. In the scientific literature a large body of work exists which describes the benefits of systems development in a virtual environment [2][3][4][5]. Recently, numerous works describe simulation environments for embedded systems. In [6] a set of tools called Embedded System Environment (ESE) is presented, especially developed for multi-processor embedded systems, based on SystemC transaction level models. The importance of virtual platforms unifying functional and robustness analysis of embedded designs is shown in tools as the one described in [7], where a simulation environment for the LEON3, a 32bit SPARC CPU used by the European Space Agency is presented. The model is TLM based and unifies simulation functionalities with fault injection capabilities. Another important aspect in the simulation of embedded systems is the verification of real time constraints. In [8], for example, the issues regarding the combination of TLM hardware models and real-time software models are analyzed. Available virtual prototyping tools can be specialized toward specific applications [9] or generally suited to SoC designs [10][11]. Some of these are based on SystemC [12][13], because of its ability to integrate hardware and software together in a common language, making it a very attractive prototyping language to system designers. These tools traditionally focus on system-on-chip designs, and their primary target is not the simulation of large, distributed and interconnected embedded systems. On the other hand, while network simulators for large systems are widely available [14][15][16][17], they generally approach the modeling problem from a high level of abstraction [18], lacking the possibility to model delays due to limited software computation power, interrupt latencies, hardware resources sharing. In our case, virtual prototyping simplifies the effort required to build and test a wide range of potential system configurations under estimated worst case conditions very early in the design process, potentially saving the costs of construction and testing on a very large system, but requires a good level of accuracy at the node level, especially regarding software processing delays and hardware resource capabilities. To achieve these results we improve the DESYRE simulation framework to realize a virtual prototype of the system, which is able to simulate the network resources and the control/application functionalities, including timing information. The element of novelty of this work is the combination of its dimensions (can scale to the order of thousand of embedded nodes), the level of accuracy in the description of each node (simulation of the hardware

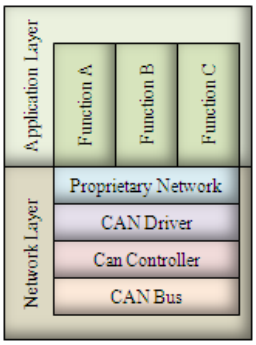


Figure 1. Layered Architecture of a physical node.

network components together with the protocol/application embedded software), the validation of the functional and timing simulation models against a physical test system.

The paper is organized as follows. In Section II we expose the details of the design problem, in Section III we describe the approach followed to simplify the modeling problem, based on the creation of a hierarchy of layers. In Section IV the DESYRE simulation framework, used to model all the components of the systems, is presented. Section V describes the back-annotation feature implemented in DESYRE for timing simulations. In Sections VI test cases, built both physically and virtually, are executed to validate the simulation model. Section VII reports the design space exploration analysis on system scaling. Finally Section VIII summarizes the obtained results.

II. Problem Formulation

The elevator system in this paper we are referring to is highly scalable for being applied to a large variety of building configurations. To reduce cost and materials, dedicated point-to-point links are replaced with a shared communication bus. As a consequence, the system presents uncertainty in its communication performance due to scalability and adoption of shared resources.

A communication bus which is commonly used throughout the transportation industry including elevator systems is the Controller Area Network (CAN) bus [19]. This bus is used for inter-node communication on which only one node can transmit at any given time, causing variability in the end-to-end latency of messages. Much work is done focusing on the timing characterization of periodic CAN messages in [20], [21]. However, the Otis Elevator system which is being modeled contains a mixture of periodic and event triggered messages. The infrequent, soft-deadline nature of the latter makes possible to share network resources to an extent which is not previously realized. However, the event-based messaging scheme comes with the disadvantage of increased complexity and processing time with a potentially serious impact to the delivery of other time critical messages. In [24] mechanisms are proposed to mitigate this performance decrement but these are not achieved without additional costs in terms of processing and bandwidth. Due to the large range of conceivable configurations, the designer has the difficult job of ensuring that the same timing requirements are met whether there are ten or ten-thousand ECUs in the system. Traditionally, these systems must be constructed and tested before any performance metrics can be collected to determine whether or not the timing requirements are satisfied. Clearly, this can be very costly and time consuming due to the size and number of possible configurations.

III. Design and Simulation Methodology

We use a Platform-Based Design (PBD) [23] approach to develop a set of SystemC-based simulation models which can be composed into a virtual prototype of our complete elevator system. Each node of our elevator system model respects the architecture of a physical node and is composed of two primary layers, (a) the application behavior layer and (b) the network communication layer. These layers are present across all physical nodes in the system, and are illustrated in Figure 1. PBD adoption allows the verification of the functionality of the system independently of the network layer. This is achieved by the simulation of the application layer only with the assumption of an ideal network. The ideal network is then refined into the different layers and the physical topology without any needs to change the application interface. For each physical node the functionalities are partitioned between software and hardware. We chose to model the functionalities implemented by software layers (Application and Proprietary Network protocol) at behavioral level, while hardware functionalities (CAN Bus, CAN Controller and CAN Driver) are modeled accurately and include exact timing computation. Latencies due to software are introduced in the simulation recurring to annotation of computational delays obtained by measures from the test system described in Section V.

IV. DESYRE: A Simulation Framework Using SystemC

The models are developed in the DESYRE[1]framework, a SystemC-based virtual prototyping environment supporting, among others, the PBD methodology and developed in several European projects [26], [27], and [28]. In this framework, the virtual prototype is composed of:

1) Functional components, written in SystemC, and/or automatically imported from other authoring tools, such as Simulink [24], and executing in zero time;

2) Architectural components refining at the transaction-level the communication and computation. They include a Real Time Operating System (RTOS) model and communication models (network protocols), exposing the effects of the resource sharing and constraints of the selected architectures;

3) Mapping components composing the previous two sets of models appropriately to create the virtual prototype of the system.

The models are organized in libraries belonging to a workspace, which defines the context where the simulation or the design space exploration, based on scenarios, take place.

[1] DESYRE - Design Environment for distributed Real-time Embedded SYstem (c) ALES S.r.l.

The virtual prototype is specified as a hierarchical network with a set of parameterized configuration input files, compliant to the IP-XACT format [25]. These files define and instantiate the simulation components, representing both hardware and software elements, and connecting them together to specify the model of the entire system. To facilitate the generation of the system netlists for complex designs and for the design space exploration, DESYRE provides an exploration language (EL) to describe in a more concise form, as parameter sets and parameter relations, the different configurations to be simulated. The EL specification is used to automatically generate and parameterize the IP-XACT file sets, representing the selected scenarios. The designer has the freedom to run selectively the simulation of a scenario or of all scenarios as a batch exploration. A simulation is performed in three phases (see Figure 2.):

1) Netlist elaboration and component loading;

2) Simulation run;

3) Data post-processing and visualization.

The first two phases are repeated for each of the chosen scenarios. In the elaboration and loading phase, according to the designer's netlists, the model builder of DESYRE dynamically creates in memory the system model by parsing the IP-XACT files and loading, through a component factory, the required SystemC models (compiled and present in the library). In the simulation run phase, the created system model is executed and the output traces are produced. In the data post-processing phase, traces are analyzed to aggregate and/or verify the performance and the functional data.

The dynamic technique for the model creation facilitates the design space exploration by 1) removing the need for the compilation of the different SystemC netlists (the flow is compiler free); 2) enabling the designer to quickly derive additional scenarios by parameterize the EL or IP-XACT files.

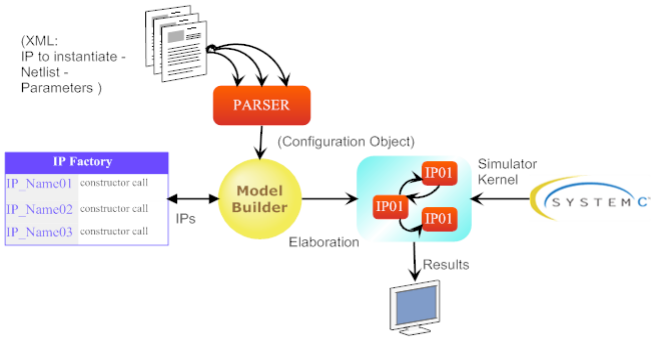


Figure 2. DESYRE Simulation Framework

During the development of this work, several optimizations to the framework are made to improve the level of automation in the process of modeling and executing the system.

V. BACK-ANNOTATION FEATURE FOR TIMING SIMULATIONS

The DESYRE framework described above is used to construct a virtual prototype of a complete elevator system according to a set of parameters including elevator system size and the specification of system inputs. Working on the accuracy of the simulator, we enhance DESYRE by adding back-annotation properties to the models, with a granularity of function calls, giving to the user the feature of setting timing parameters. In order to provide representative timing information of the actual system, we observe some important performance metrics and use these to configure the simulation models. Each software component in our model represents embedded software running on a microcontroller which requires a non-zero amount of time to execute. This time delay is measured on a test system and is captured in the models by associating a fixed computational delay for the specified action. This process of back-annotation is repeated for each transaction which can be performed by each software component. After the simulation model is annotated with estimates for computational delay, we simulate the complete elevator system, examine, and validate system-wide metrics including signal latency.

A. Test System Configuration

A scalable test system architecture is shown in Figure 3. To validate the models, a test elevator system is constructed which consists of 24 physical nodes interconnected via 3 CAN-busses. Two nodes are able to communicate on multiple CAN-busses. An identical system is constructed in our virtual prototype which we validate against the behavior and performance of the test system. We instrument the test system with a data acquisition system (DAQ) consisting of CAN-bus analyzers, logic analyzers, signal generators and signal recorders in a synchronous fashion such that a complete picture of the state of the entire system can be reconstructed at any given instant from the collected measurements. The simulation models are easily capable of providing the complete system state at all times, and we compare these to our measurements. For the measurements the following setup are used:

- *CAN-bus Analyzer,* to record the time and contents of each message;
- *Logic/State Analyzer,* to profile the execution of code on the microcontroller;
- *Signal Generator and Signal Recorder,* to provide pre-defined input signals and track system outputs.

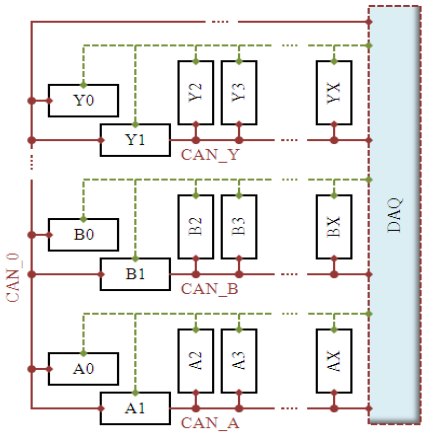


Figure 3. Scalable Test System Architecture.

B. Back-Annotation

We configure the test system described above to automatically generate inputs to the system, and collect measurements of the *computational* delay for each of the functions which implement the behavior of the network stack. The input profile is selected such that it keeps the CAN-bus traffic low to reduce the number of CAN-interrupts and opportunities for bus contention, because both can have some impact on program flow and computational delay. After measurements are collected, the appropriate function delays are extracted and combined to produce the total *software* delay for each transaction of each layer of the network stack[2]. With this data we characterize how much variability is present in each of these computational delays, and identify a single fixed value to apply to the components in the simulation model. This process is repeated for each transaction at each of the layers of the network stack. This is done so that an accurate profile of computational delays can be constructed throughout the entire network stack.

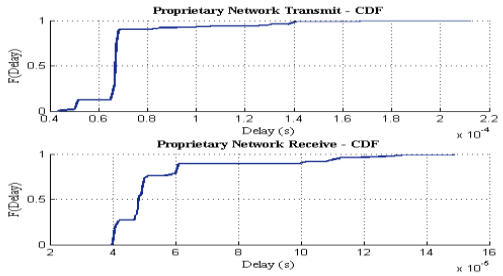


Figure 4. Measurements of Proprietary Network Computational Delay.

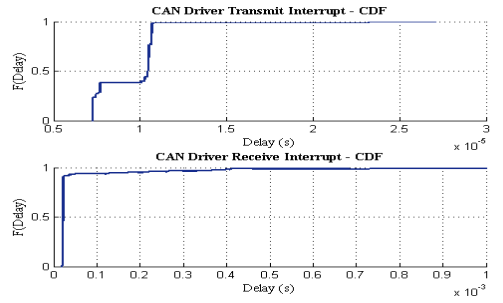


Figure 5. Measurements of CAN Driver Interrupt Computational Delay.

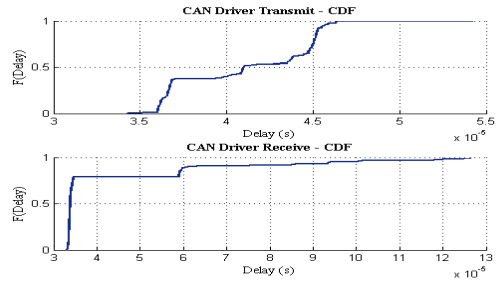


Figure 6. Measurements of CAN Driver Computational Delay.

[2] We remind that back-annotation is required only for software latencies, while network latencies due to transmissions, contention, retransmission, etc. are accurately modeled in the simulator and do not require back annotation.

The data collected at the proprietary network layer is shown in Figure 4. The data indicate that the processing delay at this component level is not constant for every transaction. We find two major reasons for this variation in delays:

- *Task Preemption*. The processing of a message is sometimes preempted by an ISR or another high priority task. When this occurs, some additional delay is added to the measurement of the processing delay. The amount of additional delay can vary widely depending on the duration of the higher priority event. For the measurements shown in Figure 4. this is observed to occur in about 1 in 10 message transactions, and accounts for the slowest 10% of messages.

- *Message Size*. Not all messages are the same size, and as a result the processing delay for each message can vary. All messages of the same size are observed to have a nearly constant processing delay. When a cumulative distribution function (CDF) is produced of all individual message processing times, we observe steps in the curve at the points corresponding to the different size messages.

We select a single value to configure the simulation model for the proprietary network transmit and receive processing delays. While selecting a value we reject the portion of messages found to be affected by preemption because the simulation model includes this behavior, and this can be counted twice if it is included here. The value selected to configure the simulation model is the slowest of the non-preempted messages, providing an approximation of the worst-case to be used as an abstraction of the processing delay of the proprietary network component during simulation. The processing delay through the CAN driver is split into two pieces which are not usually executed adjacent to each other in time: an ISR that services the hardware after each message transaction; and a conventional task which manages messages in a software queue. This non-adjacent task behavior is included in the simulation models with the implementation of a real time scheduler. Measurements are taken of both and are shown in Figure 5. and Figure 6. The same general features can be identified in the CAN Driver which is seen in the proprietary network in terms of the impact of message size and preemption on the total delay of a transaction with a couple of exceptions. The receive transaction has very little variability due to message size, and nearly all the variability comes from preemption. The transmit transaction is just the opposite in that it has very little preemption so nearly all the variability here comes from the message size. We repeat this analysis on the CAN Driver interrupts to identify the appropriate computational delays to be used during simulation. Just as with the CAN Driver receive transaction, the duration of the CAN Driver interrupt receive does not significantly depend on the size of the message. These interrupts are very rarely preempted, so the variations are entirely due to the message size or number of messages to process. The procedure for selecting the fixed values for the processing delay of the CAN driver component in the simulation model is also similar to that which is used for the proprietary network. The values selected are the largest of those that represent a single message transaction. This again provides an approximation of the worst-case to be used as an abstraction of the processing delay of the CAN driver component during simulation.

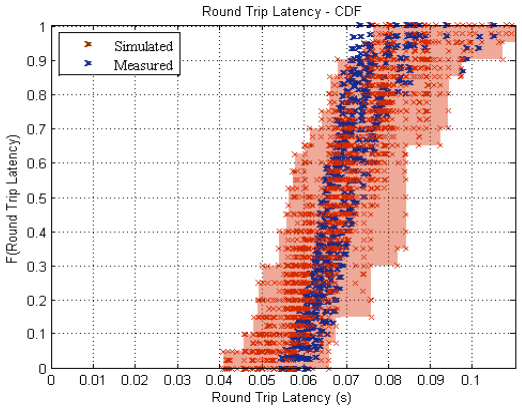


Figure 7. Message Latency Comparison for High Load Configuration.

VI. System Validation

After the model is annotated with estimates for computational delay, we can proceed to simulate the complete elevator system to examine and validate system-wide performance metrics in order to assess the accuracy of our simulation model and provide context for any system-wide simulation results which are generated. In this section, we examine the performance of the test system and compare this to the performance seen in the simulated system. This comparison is performed on the *same 24-node 3-bus test system* we use for back-annotation of the model. We define input profiles for system inputs to provide identical inputs to both the test system and the virtual system. These input profiles are drastically different from the ones used during the measurements for time back-annotation. There are both event-based and periodic messages present in the elevator system. Some nodes generate an event-based message in response to a change in an input signal, and some nodes generate event messages in response to other event messages. Our pre-defined input profiles allow us to provide a variety of repeatable inputs to the system which cause the event-based messages to be generated according to the specified profile. The variety of input profiles allows us to evaluate performance under various loading conditions. For each input profile provided to the signal generators, we collect measurements from the CAN-bus analyzers, logic analyzers, and signal recorders to reconstruct message latency, bus utilization, and queue utilization. Message latency is ultimately dependent on several other system states, including task execution state, bus utilization and queue utilization. We examine the latency of round trip messages which are configured such that they have to traverse two CAN busses. Once received on the opposite end, these messages are processed by a task which then forms a new message as a response that must also traverse the same two busses and return to the node that generated the original message. For round-trip messages, the message latency is the total time from when the first message is transmitted until the last message is received. The latency of a particular message varies each time a message is transmitted due to many factors including network delay and processing delay, so it is useful to examine the statistical properties of a set of messages as opposed to any individual message. Ten measurements are collected with a given input profile, and significant variability seen between each set of measurements. This variability is caused by the variation in task activation times due to the asynchronous behavior of tasks on multiple ECUs in the system. In order to take this asynchronous task behavior into account in the simulation, a range of simulations is run which varies relevant task activation offsets to cover the range of possible operation in the test system.

Table I. Message Latency Comparison

Method	Message Latency		
	Minimum (ms)	*Maximum (ms)*	*Average (ms)*
Simulation	40.6	115.1	69.6
Measurement	54.6	104.7	68.9

The simulation are found to cover a wider range than is observed during measurement, as reported in Table I. Therefore, the simulation model is capable of identifying more extreme message latency values than can easily be measured on a test system. The average is expected to be slightly higher in the simulation because the worst-case computational delay is back-annotated to the simulation model. This effect is then multiplied as each message encounters the worst case processing delay several times while in transit, which ultimately can be seen as a higher average latency. Each measurement and simulation run is combined into a CDF, and all are plotted together in Figure 7.

VII. Design Space Exploration

We improve the DESYRE virtual prototype to automatically explore potential elevator system configurations, and identify what, if any, are the limits of scalability, offering valuable information to the system designers. Simulations are performed on systems of various sizes and configurations, sometimes containing as many as 2400 nodes. In this section we describe an example in which we explore the scalability of the system along two dimensions, system size and input event frequency. We start with a small system configuration and gradually increase the size by adding additional elevator cars. We implement the process of automatically scaling the system, such as configuring, building, and running the system, with respect to some defined patterns. For each instantiation of a given size we examine the overall system behavior as we increase the average frequency of events across all system inputs. Events are applied to all system inputs independently, with timing between events determined by a uniform distribution. At some point during this set of tests, we expect to find that the system is overloaded, and no longer behaves according to specifications, indicating that a scaling limitation for the given system architecture is identified. As expected, this exploration identifies a scaling limit which is caused by the exhaustion of available communication resources on the CAN bus. The amount of information which must be carried over the network increases as the number of nodes on that network increases, and the CAN bus eventually becomes fully loaded. As the bus utilization reaches this limit, message queues begin to accumulate significant numbers of messages, and eventually run out of space. This represents a scaling limit due to the fact that messages are lost and correct system behavior cannot be maintained. The average event rate for the entire system and for each individual input is shown against system size in Figure 8. and Figure 9. In these plots the system behaves correctly

everywhere below the indicated line with no events or messages lost. For example, Figure 8. shows that the system behaves correctly considering a 5-floor 4-car scenario if each node transmits less than about 33events/sec; in case of a 20floors 4-cars configuration, the maximum event rate per input drops to 4events/sec. Figure 9. shows instead the maximum *overall* number of events transmitted on the system, obtained summing the events produced by each node. The plot has a maximum, because increasing the number of cars initially produces an increment in the number of nodes (transmitters) but also in bus resources (dedicated CAN lines). However, for systems with more than 4 cars, in these examples, the CAN lines shared between cars start saturating and become the bottleneck.

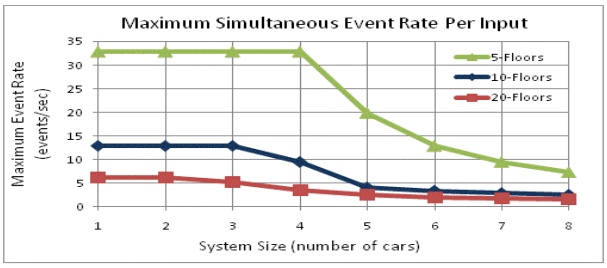


Figure 8. Exploration of Event Frequency and System Size

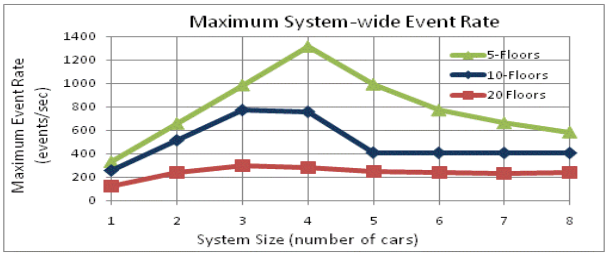


Figure 9. Exploration of Event Frequency and System Size

VIII. CONCLUSION

We have developed a virtual prototype of a scalable elevator system along with the required analysis tools in DESYRE. The behavior and timings given by the simulations are validated against a test system to ensure that the models include sufficient fidelity and the selected levels of abstraction are appropriate. The simulation models are capable of discovering best and worst-case behaviors which are not easily produced in the test system. A design space exploration analysis has reported significant results regarding the correct behavior when system scales up. Ultimately, the DESYRE simulator helps system designers to perform automatic design space exploration by identifying the characteristics of scalability prior to implementation, and ensuring that the set of expected system configurations is compliant with timing requirements.

REFERENCES

[1] OTIS Elevator Company, website http://www.otisworldwide.com

[2] G. D. Micheli, R. Ernst, and W.Wolf, "Readings in Hardware/Software codesign". San Francisco: Morgan Kaufmann publishers, Academic Press, 2002. ISBN 1-55860-702-1.

[3] D. Gajski, S. Narayan, F. Vahid and J. Gong "Specification and Design of Embedded Systems". Englewood Cliffs, NJ: Prentice-Hall, 1994.

[4] E.M. Petriu, M. Cordea, D.C. Petriu, "Virtual prototyping tools for electronic design automation", Instrumentation & Measurement Magazine, IEEE, Jun 1999, Volume 2 , Issue 2, p. 28, ISSN : 1094-6969.

[5] T. Borgstrom, E. Haritan, R. Wilson, D. Abada, R. Chandra, C. Cruse, A. Dauman, O. Mielo, A. Nohl "System prototypes: Virtual, hardware or hybrid?", Design Automation Conference, 46th ACM/IEEE, San Francisco, CA, 26-31 July 2009.

[6] S. Abdi, I. Yonghyun Hwang Lochi Yu Hansu Cho Viskic, D.D Gajski "Embedded system environment: A framework for TLM-based design and prototyping", Rapid System Prototyping (RSP), 2010 21st, IEEE International Symposium on, Fairfax, VA, 8-11 June 2010 .

[7] A. Silva, S. Sanchez "LEON3 ViP: A Virtual Platform with Fault Injection Capabilities", Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on, p. 813, Lille, 1-3 Sept. 2010 .

[8] N. Ke Yu Audsley "Combining Behavioural Real-time Software Modelling with the OSCI TLM-2.0 Communication Standard", Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on, p. 1825, Bradford, June 29 2010-July 1 2010.

[9] A. Zhonglei Wang Herkersdorf, W. Haberl, M. Wechs, "SysCOLA: A framework for co-development of automotive software and system platform", Design Automation Conference, DAC '09, 46th ACM/IEEE, p. 37 San Francisco, CA, 26-31 July 2009.

[10] Cadence Design Systems, System Design and Verification products, http://www.cadence.com

[11] Synopsys, System-Level Design Solutions products, http://www.synopsys.com

[12] Open SystemC Initiative, http://www.systemc.org

[13] D. Black, J. Donavan, *SystemC: From the Ground Up.* Springer Science+Business Media, Inc. 2004

[14] NS-2 Network Simulator, available online: http://www.isi.edu/nsnam/ns/

[15] OMNeT++, Network Simulation Framework, available online: http://www.omnetpp.org/

[16] OPNET Network Simulator, available online: http://www.opnet.com

[17] NetSim Network Simulator, available online: http://tetcos.com/software.html

[18] E. Weingartner, H. vom Lehn, K. Wehrle "A Performance Comparison of Recent Network Simulators", Communications 2009, IEEE International Conference on, pp. 1-5, Dresden, 14-18 June 2009

[19] Robert Bosch GmbH, CAN Specification, 1991.

[20] K. Tindell, A. Burns, and A.J. Wellings, "Calculating Controller Area Network (CAN) Message Response Times", *Control Eng. Practice*, 3(8), 1995, pp. 1163-1169.

[21] R.I. Davis, A. Burns, R.J. Bril, and J.J. Lukkien. "Controller Area Network (CAN) Schedulability Analysis: Refuted, Revis-ited and Revised". *Real-Time Systems*, Volume 35, Number 3, pp 239-272. April 2007.

[22] K. M. Zuberi and K. G. Shin, "Non-preemptive scheduling of messages on Controller Area Network for real-time control applications," in *Proc. Real-Time Technology and Applications Symposium*, pp.240–249,May 1995.

[23] L. Carloni, F. D. Bernardinis, C. Pinello, A. Sangiovanni-Vincentelli, and M. Sgroi. Platform-based design for embedded systems. In *The Embedded Systems Handbook.* CRC Press, 2005.

[24] Simulink © The Mathworks, www.mathworks.com

[25] The SPIRIT Consortium's ESL-based IP-XACT 1.4 specification. [Online]. Available: http://www.spiritconsortium.org/.

[26] HYCON Project. [Online]. Available: http://www.ist-hycon.org/

[27] SPEEDS IST European Project. IP Contract No. 033471. [Online]. Available: http://www.speeds.eu.com.

[28] RI-MACS - Radically Innovative Mechatronics and Advanced Control Systems. [Online]. Available: http://www.ist-world.org/ProjectDetails.aspx?ProjectId=058a4a31cf604869ba0825e07417a065

The Design and Implementation of a Simulator for Switched Ethernet Networks*

Mohammad Ashjaei, Moris Behnam, Thomas Nolte
Mälardalen University, Västerås, Sweden
{mohammad.ashjaei, moris.behnam, thomas.nolte}@mdh.se

***Abstract*—In the context of Switched Ethernet, the Flexible Time-Triggered Switched Ethernet protocol (FTT-SE) was proposed to overcome the limitations and related problems of using COTS switches in real-time networks, such as overflow of switch queues due to uncontrolled arrival of packets. Although the FTT-SE protocol has been validated by several experiments on real applications, evaluation of different architectures as well as evaluation of large scale networks is not straightforward. Therefore, a simulator to evaluate different network architectures based on the FTT-SE protocol is useful. In this paper we present such a simulator. We address the extended FTT-SE protocol using multiple switches and we present a modular simulator based on Simulink/Matlab that allows us to visualize message transmissions and to evaluate end-to-end delay bounds of messages.**

I. Introduction

Recently, there has been a growing interest in using the Switched Ethernet technology even for hard real-time distributed systems as it provides means to improve global throughput compared with other technologies. Moreover, Switched Ethernet also provides traffic isolation and it eliminates the impact of the non-determinism due to CSMA/CD arbitration that the original Ethernet was suffering from. Among several different switch types that have been proposed to support real-time traffic communication, Commercial Off The Shelf (COTS) technologies are becoming more attractive as they reduce the development costs and simplify the maintenance process compared with solutions that use dedicated switches. However, using COTS switches in real-time applications is challenging due to the following limitations; the size of the memory of COTS Ethernet switches is limited, hence it may not be possible to buffer unsynchronized simultaneous traffic from different sources which may cause packet drops that in turn affects the timeliness behavior of the network. In addition, the COTS Ethernet switches have a limited number of priorities to schedule the traffic inside switches.

One solution for the mentioned problem with respect to the COTS switch is to use a master/slave approach where one certain node will be responsible for control over the traffic communication across the network and thereby guaranteeing the real-time requirements of the traffic. In this paper, we focus on the FTT-SE protocol [1] that uses the master/slave approach to enforce global coordination among streams by using a dedicated node called the master, thus controlling the load submitted to the switch at each instant in time and thereby avoiding the potential queue overflow problem. Moreover, the FTT-SE protocol handles different traffic types including real-time periodic, real-time aperiodic and non-real-time messages by defining a specific bandwidth for each type of message. This protocol was investigated and validated by several experimental results in [1]. Moreover, a method is proposed in [2] to deal with the scalability of the protocol, using multiple switches along with multiple master nodes.

In this paper, we focus on the design and implementation of a modular simulator that can be used to simulate different design options of the FTT-SE protocol for both small and large scale networks. This simulator is based on Simulink/Matlab which makes it modular and allows us to create several models according to arbitrary architectures based on the protocol.

The rest of the paper is organized in the following manner. The next section discusses some related work on modeling and simulation of protocols, Section III describes the FTT-SE protocol and the extended solution. Then, Section IV presents the simulator design while Section V validate the simulator using some experiments. Finally, Section VI concludes the paper and presents the future work.

II. Related work

Several techniques have been proposed to model and simulate the Ethernet protocol using different tools and modeling algorithms. In [3], models are proposed for nodes, switches and traffic according to the Switched Ethernet protocol. Moreover, an evaluation is performed to validate the performance of the modeling method by comparing the simulation results with the collected data from a specific network application.

In the area of embedded avionics networks, a simulation model considering the Avionics Full Duplex Switched Ethernet (AFDX) is proposed in [4]. The end systems (nodes), switch, different queues for switch and end systems, and the measurement unit were modeled. Moreover, the validation of the modeling algorithm is performed for the specific architecture and the performance of that is compared with the results supplemented by Network Calculus. However, the simulator was developed only for a particular application and it is not implemented as a general simulator.

Furthermore, a simulation algorithm was proposed in [5] to evaluate the end-to-end upper bound delay in AFDX networks. Finding an upper bound end-to-end delay for each message

*This work is supported by the Swedish Foundation for Strategic Research, via Mälardalen Real-time Research Center (MRTC) at Mälardalen University.

using simulation requires us to investigate a huge number of possible scenarios. Thus, an approach to reduce the number of possible scenarios was proposed in [5].

In addition, different network simulation systems were designed based on available simulation tools. For instance, a network simulator system for AFDX networks was designed and implemented in [6], in which Network Simulation (NS2) as a tool to simulate TCP, routing over wired and wireless networks, was considered for the main platform, however, NS2 supports limited protocols.

Furthermore, there are many tools which have been developed for network simulation, such as TrueTime [7], OMNET++ [8] and OPNET [9]. TrueTime is a toolbox developed for Simulink/ Matlab. The switched Ethernet protocol as a network block has been supported by the TrueTime toolbox. However, adding new protocols such as FTT-SE need a lot of modifications and changes on the kernel of the tool which is not easy. Moreover, the output results that can be generated from the TrueTime blocks are limited and they need to be modified to allow for calculation of response times of messages. Another tool called OPNET is used to evaluate the performance of a network, specially for evaluation of Internet, however, this tool is a commercial tool. OMNET++ is another component-based and modular simulator which is mainly used for sensor networks, internet protocols and performance modeling. As a result, neither of them can be used directly to include the FTT-SE protocol.

III. FTT-SE BASICS

The FTT-SE protocol [1] is a real-time communication protocol that combines the master/slave technique with the Flexible Time-Triggered (FTT) paradigm. A dedicated node, called the master node is used to control the traffic in the network by broadcasting a specific message called the Trigger Message (TM). The master node schedules the ready messages according to an on-line scheduling policy, and encodes the scheduled messages into the TM. The scheduling is performed every predefined time interval called an Elementary Cycle (EC), in which the master broadcasts the TM to all slave nodes at the beginning of each EC. Then, the slave nodes receive the TM, encode it and send the scheduled messages for transmission in the current EC.

According to the FTT-SE protocol, the data transmission bandwidth in each EC is divided into two sub-bandwidths (windows) to handle synchronous (periodic) traffic within the Synchronous Window (SW) and asynchronous (aperiodic) traffic, within the Asynchronous Window (AW), as depicted in Figure 1. The time that the slave nodes need to decode the TM is called the turn around time. Moreover, the input and output ports of the switches are called the uplinks and the downlinks respectively.

Furthermore, to handle the asynchronous traffic, each slave node sends a request message, which is called signaling message (SIG), to the master node whenever an asynchronous message is activated. The master node then schedules the asynchronous traffic for upcoming ECs [10]. As illustrated

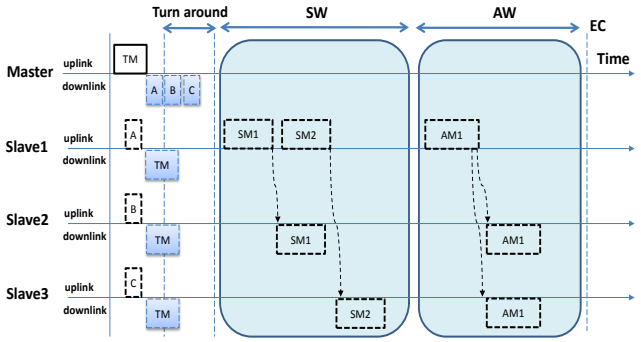


Fig. 1. The FTT-SE Elementary Cycle

in Figure 1, A, B and C are the aperiodic requests from the slave nodes. Moreover, aperiodic messages can be activated at any time during the EC. The worst case scenario occurs when an aperiodic message is activated exactly after a request has been sent to the master node. In such a scenario, the aperiodic signaling request will be sent to the master in the next EC and the master node will schedule that request at earliest in the following EC, i.e. within three ECs.

The scalability of the FTT-SE protocol using multiple switches was investigated in [11] and [2] based on two approaches. In both solutions, multiple switches are connected together directly forming a tree shaped topology.

In the first solution, a single master node is used to coordinate the traffic transmission in the network. The bandwidth assigned for synchronous and asynchronous traffic is similar to the single switch FTT-SE protocol as depicted in Figure 1. Moreover, the TM is generated and broadcasted to all nodes in the network. Also, to deal with asynchronous messages, slave nodes send the request messages to the master node.

In the second approach, a network architecture consisting of multiple switches with a master node connected to each switch is considered. An example of such an architecture is depicted in Figure 2, where the switch SW1, the master node M1 and nodes A and B are grouped into one sub-network (SN1). The sub-network SN1 is a parent sub-network for SN2 and SN3. Moreover, a cluster is defined such that it contains all sub-networks with the same parent sub-network. For instance, in Figure 2, SN4 and SN5 are grouped as one cluster for which SN2 is the parent sub-network.

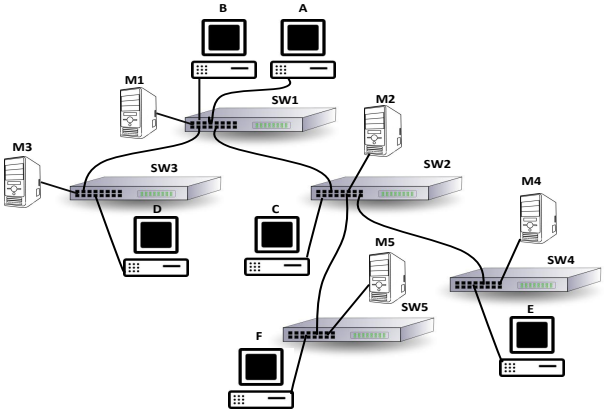


Fig. 2. An example of network

To handle the traffic in such a network, two categories of traffic are defined as local and global. The traffic which is transmitted between the nodes inside one sub-network is called local, otherwise the traffic is called global. To handle the synchronous and asynchronous traffic, the data transmission within an EC is divided among the traffic types as depicted in Figure 3, where SW is the synchronous window and AW is the asynchronous window. The local synchronous and asynchronous traffic handling is carried out similar to the single switch FTT-SE protocol within their specified bandwidth. However, all master nodes schedule all global messages in the network in parallel, based on the allocated bandwidth for such types of traffic. To schedule the global asynchronous messages, the global asynchronous window is divided per cluster and the parent master node of the cluster is responsible to schedule the global aperiodic messages inside that cluster.

Furthermore, the ECs of all master nodes are time synchronized using a particular message which is called the Global Trigger Message (GTM). The root master sends the GTM to all master nodes and they will wait to receive this message before broadcasting their local TM.

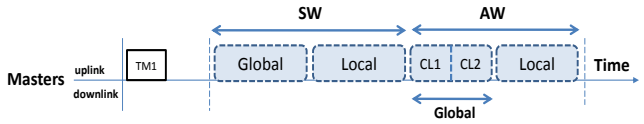


Fig. 3. EC considering local and global traffic

IV. Simulator Design

Using Simulink/Matlab we have developed a simulator to evaluate the timing behavior of the messages in a network based on the FTT-SE protocol for small and large scale networks. We have used Simulink/Matlab due to its modular features along with custom blocks and graphical interfaces. The core of the simulator is a cycle-based which starts by sending TM to the slaves, i.e. the simulator is designed in several states such that each state is allowed to execute only after finishing of the previous state. However, the master block keeps track of the EC duration as well. We have developed three basic models using the S-Function block in Simulink to simulate the functionality of the master node, slave nodes and switch models respectively. These blocks are stored in a Simulink library file. To simulate the parallel execution of blocks in a model, we have divided each EC into 100 time slots and in each of them all the functions are executed. The number of time slots shows the resolution of simulation which is not fixed and can be changed in the configuration of the simulator. However, by increasing the number of time slots, the simulation time will increase due to the number of functions that need to be execute in each time slot. On the other hand, decreasing the time slots number can affect on the accuracy of the results. Therefore, we have chosen 100 time slots in this trade-off as an engineering experiment. Note that, the function of blocks in Simulink executes in sequential order which is automatically specified by Matlab in advance. Therefore, the input data of each block is guaranteed to be available before the execution of that block.

A. Ready Queues Management

For the case of multiple master nodes, each master contains four ready queues to support local and global periodic as well as local and global aperiodic messages. Scheduling of global messages is performed in parallel in all master nodes at the same time. The ready queues are sorted based on the priority of messages in which the highest priority messages are inserted at the head of the queues, we used the Fixed Priority/Highest Priority First Scheduling Policy for on-line scheduling in the simulator. Messages with the same priority are sorted in the queues based on the First Come First Serve (FCFS) policy for local messages, whereas for global messages, messages having the same priority are sorted based on the id number of messages.

For management of the ready queues, we have developed three functions to handle queue updating before scheduling the messages by master nodes. These functions, which are implemented in Matlab m-files, are the following:

Get head message. This function returns the first message in the ready queue which is always the highest priority message among all messages in the ready queue in this simulation.

Remove a message. If the scheduler checks a message and it selects that message to be transmitted in the current EC, the message should be removed from the ready queue. Therefore, this function removes a message defined by its id together with the ready queue in which the message is residing. The output of this function is the updated ready queue sorted according to the priorities of all messages in the queue.

Insert a message and sort in ascending format. Whenever a message becomes ready, it should be inserted in the correct ready queue, which is performed by this function. This function inserts a message in a queue according to its priority and it re-sorts the queue according to the priorities of messages in which the highest priority message is assigned at the head of the queue.

For the single master case, two queues are used to schedule both synchronous and asynchronous messages, one specific for each type of messages. Therefore, the queue management functions are applicable for these two ready queues.

B. Master Block Design

The master block is divided into two sub-blocks dealing with sender and receiver functions. We have defined an array structure for the master node to store its variables and parameters. In the solution with multiple master nodes, each master node in the network may have different parameters depending on its local configuration such as local message numbers and local slave node properties. All master blocks in the model are connected to a single m-file function which is distinguished with a mask block parameter number.

For each master input (receiver) and output (sender) blocks two separate functions are implemented, however the master function is developed based on a state flow that each of them should run in order. The master function has three states which are depicted in Figure 4. Each state, is allowed to run when the previous state has executed only. The first state is broadcasting

the TM to all slaves. The next state is receiving aperiodic requests from the slave nodes during the TM window. The last state is performing the scheduling function for all kind of messages, including local and global aperiodic messages and generating a TM for the upcoming EC. State 1 and 3 are executed in the master sender function, whereas state 2 is executed in the receiver function. For input and output signals, we implemented separate scope output functions due to the flexibility of the graphical interface.

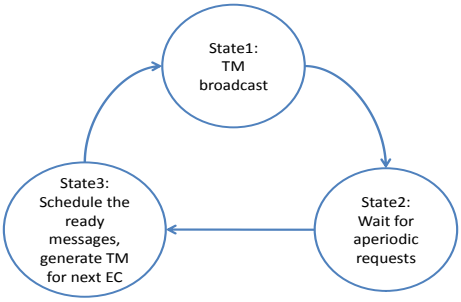


Fig. 4. Master function state flow

In state 2 of the master function, which is executed in the master receiver sub-block, the master polls the aperiodic requests from the slave nodes. This state finishes and moves to the next state when all signal messages are received. The aperiodic requests which are received during this state are from local slave nodes or from children slaves for global aperiodic requests. In both cases, the requests are stored and the master schedules them for the next EC. The local aperiodic scheduled messages are encoded inside the TM along with other periodic messages, however the global scheduled aperiodic messages are encoded in a different trigger message called the asynchTM, which is sent to children slave nodes directly.

In state 3, the master function looks up the message structure and checks whether any periodic message becomes ready. The scheduler inserts all ready messages, including aperiodic messages, which are requested from the slaves into the related ready queues. Moreover, the ready time of each message is stored in the related variable of message. Consequently, scheduler function checks bandwidth for the ready message transmission according to their destination and rout. The scheduled messages are encoded inside the TM to be sent at the beginning of the next EC.

To support the single master FTT-SE network, the master block is developed such that it schedules the traffic and broadcasts the TM to the entire network. Moreover, each slave node sends the aperiodic request to the master node which is connected to the top switch.

C. Switch Model Design

Similar to the master block, the switch is modeled with a Matlab S-Function associated using a particular m-file. Four kinds of connections are defined for the switch models i) the master connection identified in port 1, ii) the parent switch connection is dedicated to port 2, iii) two children connections for the children sub-networks as port 3 and 4, and finally iv) five connections for the slave nodes. Therefore, 9 ports are assumed for switches in this version of the simulator. Moreover, for each input and output link a specific buffer is assigned to store receiving and sending data. The buffers are sorted according to the First In First Out (FIFO) policy. The general structure of the switch model is to poll the input data and to process the destination address of them, and in turn, to insert into the related output buffer. The switch parameters including the input and the output buffers are stored in an array structure which is shown in Table I.

switch structure	description
port_nbr	The number of ports for switches.
inBuffer	The input buffer for all input ports individually.
outBuffer	The output buffer for all output ports individually.

TABLE I
THE SWITCH STRUCTURE

D. Slave Block Design

Similar to the master block, the slave block is divided into two sub-blocks denoted sender and receiver sub-blocks. The slave function is executed based on the state flow which indicates the current state of each slave node. The slave function composes of four individual states started by the TM reception. The state flow of the slave function is depicted in Figure 5.

After receiving the asynchTM and TM messages from the parent master and the main master respectively, the slave checks if any local aperiodic messages are ready to be transmitted. The same check is performed for the global aperiodic messages. For aperiodic messages, the sporadic model is used to model this type of traffic in which the minimum inter-arrival time is defined for each message, however in this simulator a dynamic activation for aperiodic messages is developed to simulate the unpredictable arrival time of aperiodic messages. Moreover, the aperiodic message may become ready at anytime during the EC window. To simulate this behavior of aperiodic messages, we assume the worst case in which the message always becomes ready after the TM broadcasting window.

The third state of the slave function, which is executed in the sender block, is decoding the TM and transmitting the messages which are scheduled by master node. This state includes all local/global periodic and aperiodic messages. However, the message transmission starts with local and global periodic messages and continues with local and global aperiodic messages.

After sending the scheduled messages, the last state is to wait for message receiving. The slaves read their inputs until the EC time window is finished. When a message is received from the slave node, the receiving time is stored in the related message variable. The time interval between the ready time and the receiving time of the messages shows the end-to-end transmission time. For setting the receiving time, the store-and-forward switch delay and order of messages are considered to simulate as accurately as possible. Since the bandwidth capacity was checked in the scheduler, then all scheduled

messages should be received in the current EC without any deadlines being missed. In case of a deadline miss or a failure in receiving of message, the output report of the simulator will show it.

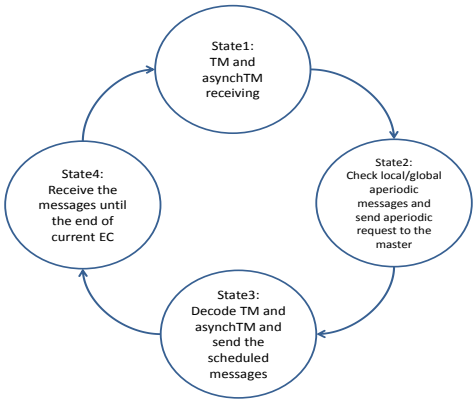


Fig. 5. Slave function state flow

For generating the scope output signals to cover both receiving and sending messages, the scope functions store the messages which are sent and received. The messages are scoped according to their transmitting time. Each message is indicated by its identification number which is unique in the entire network.

E. Settings and Report

In order to set the configuration of network example, such as the EC size and the bandwidth allocations, a database which is developed in Matlab m-file is prepared. For each network model, different configuration can be determined to assess the performance of the example in different bandwidth assignments.

Moreover, to present the end-to-end delay of messages, after finishing the simulation, a function is developed to generate the output reports. These reports provide information including the network parameters such as number of switches and masters, EC configuration and end-to-end delays including minimum, average and maximum that has been measured during the simulation.

V. Examples

In this section, we present an example of a network that consists of 5 switches (sub-networks) as depicted in the Simulink model in Figure 6. This network composes of 16 slave nodes in which 2-4 nodes are connected to each sub-network. The network parameters for this example are $EC = 4ms$, $TM = 12\mu s$, $SIG = 6\mu s$ and the transmission speed of the Ethernet network is considered as $100Mbps$ and 40 messages are generated randomly. The Fixed Priority Scheduling Policy is assumed in this example and the priority of messages is selected according to the Rate Monotonic priority assignment. Note that, the simulator can support higher amount of messages if all messages are schedulable (meet their deadlines) in the network architecture. In this example we have experimented 40 messages to present the results considering the space limit.

A. Multiple Masters Network

In the multiple masters architecture, each switch is connected to a single master, i.e., five master nodes are created in Simulink, for illustration purpose we explain the root sub-network model in Figure 7. Moreover, the transmission bandwidth in each EC is divided as follows. The synchronous local and global scheduling windows are selected to have $1ms$ equally, the asynchronous local scheduling window is $800\mu s$ and finally the asynchronous global scheduling window is $700\mu s$. In the example, the network is composed of two clusters and the bandwidth of the asynchronous global scheduling window is further divided equally among them, i.e., $350\mu s$.

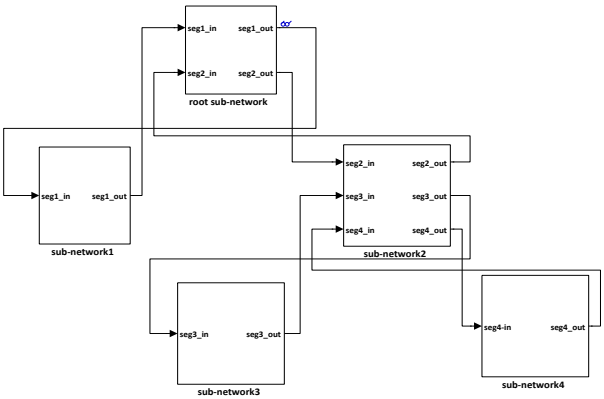


Fig. 6. Evaluation example

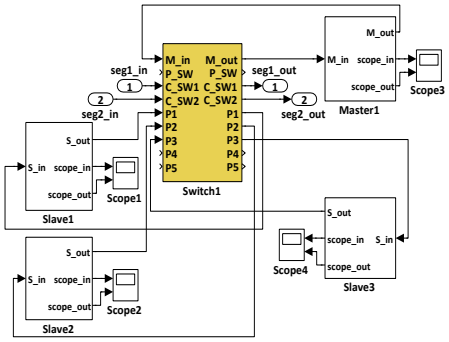


Fig. 7. Root sub-network model

The simulation for this example is performed for time duration of 500 ECs of simulation time. To visualize the message transmission, an ordinary Scope block of Simulink is attached to the respective scope ports of the master and slave blocks. For instance, the messages transmitted and received by slave node 6 in the root sub-network are illustrated in Figure 8, where the x-axis presents time and y-axis shows the message id. In this example the messages $m21$, $m23$ and $m33$, which are transmitted from slave node 6, are periodic with the priority equal to 4, 4 and 1 respectively (higher numbers represent higher priority). Also, this slave node receives the messages $m20$ and $m22$ which are periodic as well. The

message id	Min. RT	Avg. RT	Max. RT
20	2	2	2
21	2	2	2
22	2	2	2
23	2	2	2
33	2	2	2

TABLE II
MESSAGE RESPONSE TIMES

transmission time of each message is depicted in the scope with respect to the declaration of the messages. Moreover, the end-to-end delay of all messages in the model can be reported after the simulation time is finished. The minimum, average and maximum response time, which are measured during the simulation, for the mentioned messages are shown in Table II (the unit used is multiples of ECs).

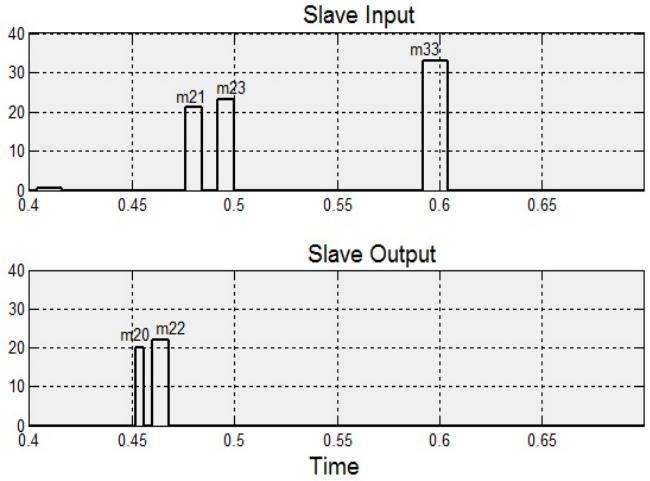


Fig. 8. Scope input/output of slave number 6 in example 1

B. Single Master Network

In this section, we present the results of applying the approach of a single master on the example presented in the previous section. The master node is connected to the Switch 1. The synchronous window and the asynchronous window are selected to have 2*ms* and 1.5*ms* respectively. Similar to the previous example, the simulator is executed for 500 ECs of simulation time. Figure 9 shows the message transmissions in the slave number 7.

VI. CONCLUSION AND FUTURE WORK

In this paper, we addressed the FTT-SE protocol and the extended FTT-SE protocol to support large scale networks using multiple switches along with multiple master nodes. Moreover, we presented the design of a simulator based on Simulink/Matlab. The simulator contains three basic models representing master, switch and slaves, all implemented as S-Function blocks in Simulink to make the simulator modular.

Moreover, two examples consisting of five sub-networks are created in the simulator to evaluate both the FTT-SE approaches. Different output scopes of message transmissions are presented along with end-to-end delay reports using both a single master as well as multiple masters. We have presented this tool that allows us to perform detailed analysis of the protocols in a way that before needed implementation with

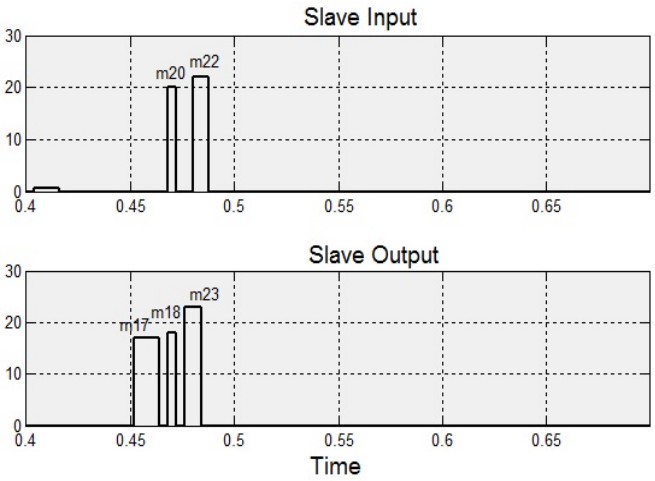


Fig. 9. Scope input/output of slave number 7 in example 2

all its complexity. This version of the simulator is designed and implemented considering some restriction assumptions, e.g. the switch model is limited to support two children sub-networks. However, the extension of the simulator is currently ongoing.

Furthermore, this tool is extensible in the sense that we can easily accommodate other Ethernet protocols for comparison, since the core of the simulator is already implemented. Moreover, we are planning on making the tool available for public as a downloadable plug in to Simulink/Matlab.

REFERENCES

[1] R. Marau, L. Almeida, and P. Pedreiras, "Enhancing real-time communication over cots ethernet switches," in *6th IEEE International Workshop on Factory Communication Systems (WFCS'06)*, June 2006.

[2] M. Ashjaei, M. Behnam, T. Nolte, L. Almeida, and R. Marau, "A compact approach to clustered master-slave ethernet networks," *9th IEEE International Workshop on Factory Communication Systems (WFCS'12)*, May 2012.

[3] Z. Huang, Y. Zhang, and H. Xiong, "Modeling and simulation of switched ethernet," in *2nd International Conference on Computer Modeling and Simulation (ICCMS'10)*, vol. 3, January 2010.

[4] H. Charara and C. Fraboul, "Modeling and simulation of an avionics full duplex switched ethernet," in *Advanced industrial conference on telecommunications/service assurance with partial and intermittent resources conference/e-learning on telecommunications workshop*, July 2005.

[5] J.-L. Scharbarg and C. Fraboul, "Simulation for end-to-end delays distribution on a switched ethernet," in *12th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'07)*, September 2007.

[6] S. Dong, Z. Xingxing, D. Lina, and H. Qiong, "The design and implementation of the afdx network simulation system," in *International Conference on Multimedia Technology (ICMT'10)*, October 2010.

[7] D. Henriksson, A. Cervin, and K.-E. Årzén, "TrueTime: Real-time control system simulation with MATLAB/Simulink," in *Proceedings of the Nordic MATLAB Conference*, Copenhagen, Denmark, October 2003.

[8] "Omnet++: Component-based c++ simulation library, available at http://www.omnetpp.org."

[9] "Opnet: Application and network performance, available at http://www.opnet.com."

[10] R. Marau, P. Pedreiras, and L. Almeida, "Asynchronous traffic signaling over master-slave switched ethernet protocols," in *6th International Workshop on Real Time Networks (RTN'07)*, July 2007.

[11] R. Marau, M. Behnam, Z. Iqbal, P. Silva, L. Almeida, and P. Portugal, "Controlling multi-switch networks for prompt reconfiguration," in *Proc. of 9th Int. Workshop on Factory Communication Systems (WFCS'12)*, May 2012.

SoOSiM: Operating System and Programming Language Exploration

Christiaan Baaij, Jan Kuper
Computer Architecture for Embedded Systems,
Department of EEMCS, University of Twente,
Postbus 217, 7500AE Enschede, The Netherlands
Email: {c.p.r.baaij;j.kuper}@utwente.nl

Lutz Schubert
HLRS – University of Stuttgart,
Department of Intelligent Service Infrastructures,
Nobelstr. 19, D-70569 Stuttgart, Germany
Email: schubert@hlrs.de

***Abstract*—SoOSiM is a simulator developed for the purpose of exploring operating system concepts and operating system modules. The simulator provides a highly abstracted view of a computing system, consisting of computing nodes, and components that are concurrently executed on these nodes. OS modules are subsequently modelled as components that progress as a result of reacting to two types of events: messages from other components, or a system-wide tick event. Using this abstract view, a developer can quickly formalize assertions regarding the interaction between operating system modules and applications.**

We developed a methodology on top of SoOSiM that enables the precise control of the interaction between a simulated application and the operating system. Embedded languages are used to model the application once, and different interpretations of the embedded language constructs are used to observe specific aspects on application's execution. The combination of SoOSiM and embedded languages facilitates the exploration of programming language concepts and their interaction with the operating system.

I. Introduction

Simulation is a commonly used tool in the exploration of many design aspects of a system: ranging from feasibility aspects to gathering performance information. However, when tasked with the creation of new operating system concepts, and their interaction with the programmability of large-scale systems, existing simulation packages do not seem to have the right abstractions for fast design exploration [1], [2] (ref. Section IV). The work we present in this paper has been created in the context of the S(o)OS project [3]. The S(o)OS project aims to research OS concepts and specific OS modules, which aid in scalability of the complete software stack (both OS and application) on future many-core systems. One of the key concepts of S(o)OS is that only those OS modules needed by a application thread, are actually loaded into the (local) memory of a Core / CPU on which the thread will run. This execution environment differs from contemporary operating systems where every core runs a complete copy of the (monolithic) operating system.

A basic requirement that the S(o)OS project has towards any simulator, are the facilities to straightforwardly simulate the instantiation of application threads and OS modules. Aside from the fact that the S(o)OS-envisioned system will be dynamic as a result of loading OS modules on-the-fly; large-scale systems also tend to be dynamic in the sense that computing nodes can (permanently) disappear (failure), or appear (hot-swap). Hence, the simulator has to facilitate the straightforward creation and destruction of computing elements. The current need for a simulator rests mostly in formalizing the S(o)OS concept, and examining the interaction between the envisioned OS modules and the application threads. As such, being able to extract highly accurate performance figures from a simulated system is not a key requirement. It should however facilitate the ability to observe all interactions between application threads and OS modules should. Additionally, a user should able to *zoom in* on particular aspects of the behaviour of an application: such as memory access, messaging, etc.

This paper describes a new simulator, *SoOSiM* , that meets the above requirements. We elaborate on the main concepts of the simulator in Section II, and show how OS modules interact with each other, and with the simulator. Section III describes the use of embedded languages for the creation of applications running in the simulated environment. The simulation engine, the graphical user interface, and embedded language environment are all written in the functional programming language Haskell [4]; this means that all code listings in this paper also show Haskell code. Due to limitation in the number of pages, we are not be able to elaborate every Haskell notation; the code examples are intended to support the validity of the presented concepts. Section IV compares SoOSiM to existing simulation frameworks, and lists other related work. Section V enumerates our experiences with SoOSiM, and Section VI discusses potential future work.

II. Abstract System Simulator

The purpose of SoOSiM is mainly to provide a platform that allows a developer to observe the interactions between OS modules and application threads. It is for this reason that the simulated hardware is highly abstract. In SoOSiM, the hardware platform is described as a set of nodes. Each *node* represents a physical computing object: such as a core, complete CPU, memory controller, etc. Every node has a local memory of potentially infinite size. The layout and connectivity properties of the nodes are not part of the system description.

This work is supported through the S(o)OS project, sponsored by the European Commission under FP7-ICT-2009.8.1, Grant Agreement No. 248465.
SoOSiM is available on: http://hackage.haskell.org/package/SoOSiM

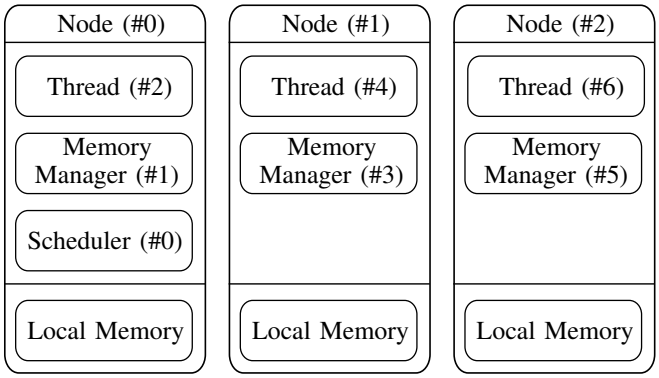


Fig. 1. Abstracted System

Each *node* hosts a set of components. A *component* represents an executable object: such as a thread, application, OS module, etc. Components communicate with each other either using direct messaging, or through the local memory of a node. Having both explicit messaging and shared memories, SoOSiM supports the two well known methods of communication. Components have a (hidden) message queue, because:

- Multiple components can send messages to the same component concurrently.
- A component can receive messages while it is waiting for a response from another component.

All components in a simulated system, even those hosted within the same node, are executed concurrently (from the component's point of view). The simulator poses no restrictions as to which components can communicate with each other, nor to which node's local memory they can read from and write to. A schematic overview of an example system can be seen in Figure 1.

The simulator progresses all components concurrently in one discrete step called a $tick$. During a *tick*, the simulator passes the content that is at the head of the message queue of each individual component. If the message queue of a component is empty, a component will be executed with a *null* message. If desired, a component can inform the simulator that it does not want to receive these *null* messages. In that case the component will not be executed by the simulator during a *tick*.

A. OS Component Descriptions

Components of the simulated system are, like the simulator core, also described in the functional programming language Haskell. This means that each component is described as a function. In case of SoOSiM, such a function is not a simple algebraic function, but a function executed within the context of the simulator. The Haskell parlance for such a computational context is a *Monad* [4, Chapter 14], the term we will use henceforth. Because the function is executed within the monad, it can have *side-effects* such as sending messages to other components, or reading the memory of a local memory. In addition, the function can be temporarily suspended at (almost) any point in the code. SoOSiM needs to be able to suspend the execution of a function so that it may emulate synchronous messaging between components, a subject we will further elaborate later on.

We describe a component as a function that, as its first argument, receives a user-defined internal state, and as its second argument a value of type $Event\ a$. The result of this function will be the (potentially updated) internal state. Values of type $Event\ a$ can either be:

- A message from another component, where 'a' represents the datatype of the content of the message.
- A *null* message.

We thus have the following type signature for a component:

$$component :: State \rightarrow Event\ a \rightarrow Sim\ State$$

The Sim annotation on the result type means that this function is executed within the simulator monad. The user-defined internal state can be used to store any information that needs to perpetuate across simulator *ticks*.

To include a component description in the simulator, the developer will have to create a so-called *instance* of the $ComponentInterface$ *type-class*. A *type-class* [4, Chapter 6] in Haskell can be compared to an interface definition as those known in object-oriented languages. An *instance* of a *type-class* is a concrete instantiation of such an interface. SoOSiM users should use a singleton datatype to uniquely label the interface description of a component. The $ComponentInterface$ consists of the following values to completely define a component:

- The datatype representing the internal state.
- The datatype of the received messages.
- The datatype of the send messages.
- The initial internal state of the component.
- The unique name of the component.
- The monadic function describing the behaviour.

We stress again that we are aiming at a high level of abstraction for the behavioural descriptions of our OS modules, where the focus is mainly on the interaction with other OS modules and application threads.

B. Interaction with the simulator

Components have several functions at their disposal to interact with the simulator and consequently interact with other components. The available functions are the following:

- $createComponent$ instantiates a new component on a specified node.
- $invoke$ sends a message to another component, and waits for the answer. Whenever a component uses this function it will be suspended by the simulator. Several simulator ticks might pass before the callee sends a response. Once the response is available the simulator resumes the execution of the calling component.
- $invokeAsync$ sends a message to another component, and registers a handler with the simulator to process the response. In contrast to $invoke$, using this function will *not* suspend the execution of the component.
- $respond$ sends a message to another component as a response to an invocation.

- *yield* informs the simulator that the component does not want to receive *null* messages.
- *readMem* performs a read at a specified address of a node's local memory.
- *writeMem* writes a value at a specified address of a node's local memory.
- The *componentLookup* function performs a lookup of the unique identifier of a component given a specified interface.

Components have a *ComponentId* that is a unique number corresponding to a specific instance of a component. The knowledge of the unique *ComponentId* of the specific instance is needed to invoke a component. To give a concrete example, using the system of Figure 1 as our context: *Thread(#6)* wants to invoke the instance of the *MemoryManager* that is running on the same Node (#2). As *Thread(#6)* was not involved with the instantiation of that OS module, it has no idea what the specific *ComponentId* of the memory manager on Node #2 is. It does however know the interface-label of the memory managers, so it can use the *componentLookup* function to find the *MemoryManager* with ID #5 that is running on Node #2.

C. Example OS Component: Memory Manager

This subsection demonstrates the use of the simulator API, taking the *Read* code-path of the memory manager module as an example. The memory manager takes care that the reads or writes of a global address end up in the correct node's local memory. As part of its internal state the memory manager keeps a lookup table. This lookup table states whether an address range belongs to the local memory of the node that hosts the memory manager, or whether that address is handled by a memory manager on another node. An entry of the lookup table has the following datatype:

```
data Entry = EntryC
  { base  :: Int
  , scope :: Int
  , scrId :: Maybe ComponentId
  }
```

The fields *base* and *scope* together describe the memory address range defined by this entry. The *srcId* tells us whether the range is hosted on the node's local memory, or whether another memory manager is responsible for the address range. If the value of *scrId* is *Nothing* the address is hosted on the node's local memory; if *srcId* has the value *Just cmpId*, the memory manager with ID *cmpId* is responsible for the address range.

Listing 1 highlights the Haskell code for the read-logic of the memory manager. Lines 1, 2, and 3 show the type signature of the function defining the behaviour of the memory manager. On line 4 we use pattern-matching, to match on a *Message* event, binding the values of the message content, and the identification of the caller, to *content* and *caller* respectively. We examine the *content* on line 4, and only continue when it is a *Read* message (indicated by the vertical bar |). If it is a *Read* message, we bind the value of the address to the name *addr*. On line 7 we lookup the address range entry which encompasses *addr*. Line 8 starts a **case**-statement discriminating on the value of the *srcId* of the entry. If the *srcId* is *Nothing* (line 9-12), we read the node's local memory using the *readMem* function, *respond* to the caller with the read value, and finally *yield* to the simulator. When the address range is handled by a *remote* memory manager (line 13-17), we *invoke* that specific memory manager module with the read request and wait for a response. We remark that many simulator cycles might pass between the invocation and the return, as the *remote* memory manager might be processing many requests. Once we receive the value from the *remote* memory manager, we *respond* to the original caller forwarding the received value.

Note that the functions *invoke* and *respond* each receive, as their first argument, the singleton-datatype that was used to label the memory manager interface. This label is used to access the *Receive* and *Send* datatype fields of the interface, and statically ensures that we only send and receive datatypes that correspond to the interface of the memory manager.

Listing 1 Read logic of the Memory Manager

```
memoryManager :: MemState                                  1
              -> Event MemCommand                          2
              -> Sim MemState                              3
memoryManager s (Message content caller)                   4
  | (Read addr) <- content                                 5
  = do                                                     6
    let entry = addressLookup s addr                       7
    case (srcId entry) of                                  8
      Nothing -> do                                        9
        addrVal <- readMem addr                            10
        respond MemoryManager caller addrVal               11
        yield s                                            12
      Just remote -> do                                    13
        response <- invoke MemoryManager                   14
                           remote content                  15
        respond MemoryManager caller response              16
        yield s                                            17
  | (Write addr val) <- content                            18
  = do                                                     19
    ...                                                    20
```

D. Simulator GUI

The state of a simulated system can be observed using the SoOSiM GUI, of which a screenshot is shown in Figure 2. The GUI allows you to run and step through a simulation at different speeds. On the screenshot we see, at the top, the toolbar controlling the simulation, in the middle, a schematic overview of the simulated system, and specific information belonging to a selected component at the bottom. Different colours are used to indicate whether a component is active, waiting for a response, or idle. The *Component Info* box shows both static and statistical information regarding a selected component. Several statistics are collected by the simulator, including the number of simulation cycles spent in a certain state (active / idle / waiting), messages sent and received, etc.

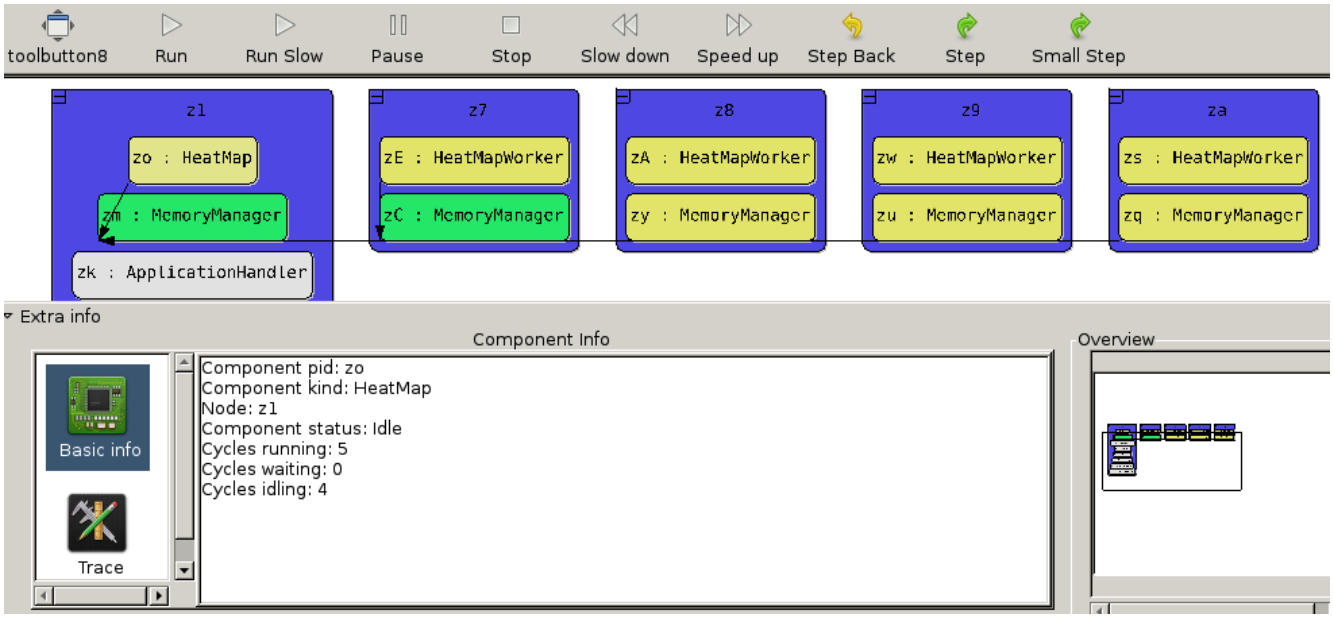


Fig. 2. Simulator GUI

These statistics can be used to roughly evaluate the performance bottlenecks in a system. For example, when OS module 'A' has mostly active cycles, and components 'B'-'Z' are mostly waiting, one can check if components 'B'-'Z' were indeed communicating with 'A'. If this happens to be the case, then 'A' is indeed a bottleneck in the system. A general rule-of-thumb for a well performing system is when OS modules have many *idle* cycles, and application threads have mostly *active* cycles.

III. Embedded Programming Environment

One of the reasons to develop SoOSiM is to observe the interaction between applications and the operating system. Additionally, we want to explore programming language concepts intended for parallel and concurrent programming, and how they impact the entire software stack. For this purpose we have developed a methodology on top of SoOSiM, that uses embedded languages to specify the applications. Our methodology consists of two important aspects:

- The use of embedded (programming) languages to define an application.
- Defining different interpretations for such an application description, allowing a developer to observe different aspects of the execution of an application.

A. Embedded Languages

An *embedded language* is a language that can be used from within another language or application. The language that is embedded is called the *object* language, and the language in which the *object* language is embedded is called the *host* language. Because the *object* language is *embedded*, the *host* language has complete control over any terms / expressions defined within this *object* language. There are multiple ways of representing embedded languages, for example as a string, which must subsequently be parsed within the *host* language.

Haskell has been used to host many kinds of embedded (domain-specific) languages [5]. The standard approach in Haskell is not to represent *object* terms as strings, but instead use data-types and functions. To make this idea more concrete, we present the recursive Fibonacci function, defined using one of our self-defined *embedded* functional languages, in Listing 2.

Listing 2 Call-by-Value Fibbonaci

```
fib :: Symantics repr ⇒ repr (IntT ↠ IntT)                    1
fib = fix $ λf →                                              2
  fun $ λn →                                                  3
    nv 0 $ λn1 →                                              4
    nv 0 $ λn2 →                                              5
    nv 0 $ λn3 →                                              6
      n1 =: n `seq`                                           7
      if_ (lt (drf n1) 2)                                     8
        1                                                     9
        (n2 =: (app f (drf n1 − 1)) `seq`                     10
         n3 =: (app f (drf n1 − 2)) `seq`                     11
         drf n2 + drf n3                                      12
        )                                                     13
```

All functions printed in **bold** are language constructs in our *embedded language*. Additionally the =: operator is also one of our *embedded* language constructs; the numeric operators and literals are also overloaded to represent embedded terms. To give some insight as to how Listing 2 represents the recursive Fibonacci function, we quickly elaborate each of the lines.

The type annotation on line 1 tells us that we have a function defined at the *object*-level (↠) with an *object*-level integer (*IntT*) as argument and an *object*-level integer (IntT) as result.

Line 2 creates a fixed-point over f, making the recursion of our embedded Fibonacci function explicit. On line 3 we define a function parameter n using the **fun** construct. We remark that we use Haskell binders to represent binders in our *embedded language*. On line 4-6 we introduce three mutable references, all having the initial integer value of 0. We assign the value of n to the mutable reference $n1$ on line 7. On line 8 we check if the dereferenced value of $n1$ is less than 2; if so we return 1 (line 9); otherwise we assign the value of the recursive call of f with $(n1 - 1)$ to $n2$, and assign the value of the recursive call of f with $(n1 - 2)$ to $n3$. We subsequently return the addition of the dereferenced variables $n2$ and $n3$.

We must confess that there is some syntactic overhead as a result of using Haskell functions and datatypes to specify the language constructs of our *embedded* language; as opposed to using a string representation. However, we have consequently saved ourselves from many implementation burdens associated with embedded languages:

- We do not have to create a parser for our language.
- We can use Haskell bindings to represent bindings in our own language, avoiding the need to deal with such *tricky* concepts as: symbol tables, free variable calculation, and capture-free substitution.
- We can use Haskell's type system to represent types in our embedded language: meaning we can use Haskell's type-checker to check expressions defined in our own embedded language.

B. Interpreting an Embedded Language

We mentioned the concept of *type-classes* when we discussed the process of including a component description in the simulator. Following the *final tagless* [6] encoding of embedded languages in Haskell, we use a type-class to define the language constructs of our mini functional language with mutable references. A partial specification of the *Symantics* (a pun on *syntax* and *semantics* [6]) type-class, defining our *embedded language*, is shown in Listing 3.

Listing 3 Embedded Language - Partial Definition. Viz. [6]

```
class Symantics repr where                                          1
  fun :: (repr a → repr b) → repr (a ↠ b)                           2
  app :: repr (a ↠ b) → repr a → repr b                             3
  drf :: repr (Ref a) → repr a                                      4
  (=:) :: repr (Ref a) → repr a → repr Void                         5
```

We read the types of our language definition constructs as follows:

- **fun** takes a *host*-level function from *object*-type a to *object*-type b ($repr\ a \rightarrow repr\ b$), and returns an *object*-level function from a to b ($a \twoheadrightarrow b$).
- **app** takes an *object*-level function from a to b, and applies this function to an *object*-term of type a, returning an *object*-term of type b.
- **drf** dereferences an *object*-term of type "reference of" a (written in Haskell as $Ref\ a$), returning an *object*-term of type a.
- $(=:)$ is operator that updates an *object*-term of type "reference of" a, with a new *object*-value of type a, returning an *object*-term of type $Void$.

To give a desired interpretation of an application described by our embedded language we simply have to implement an instance of the $Symantics$ type-class. These interpretations include pretty-printing the description, determining the size of expression, evaluating the description as if it were a normal Haskell function, etc.

In the context of this paper we are however interested in *observing* (specific parts of) the execution of an application inside the SoOSiM simulator. As a running example, we show part of an instance definition that observes the invocations of the memory manager module upon dereferencing and updating mutable references:

Listing 4 Observing Memory Access - Partial definition

```
instance Symantics Sim where                                        1
  drf x = do                                                        2
    i     ← foo x                                                   3
    mmId ← componentLookup MemoryManager                            4
    invoke MemoryManager mmId (Read i)                              5
  x =: y = do                                                       6
    i     ← foo x                                                   7
    v     ← bar y                                                   8
    mmId ← componentLookup MemoryManager                            9
    invoke MemoryManager mmId (Write i v)                           10
```

We explained earlier that the simulator *monad* (Sim) should be seen as a computational context in which a function is executed. By making our simulator monad the computational **instance** (or environment) of our embedded language definition, we can now run the applications defined with our embedded language inside the SoOSiM simulator. Most language constructs of our embedded language will be implemented in such a way that they behave like their Haskell counterpart. The constructs where we made minor adjustments are the **drf** and (=:) constructs, which now enact communication with our *Memory Manager* OS module. By using the $invoke$ function, our application descriptions are also suspended whenever they dereference or update memory locations, as they have to wait for a response from the memory manager. Using the SoOSiM GUI, we can now observe the communication patterns between the applications described in our embedded language, and our newly created OS module.

C. Further Extensions and Interpretations

The use cases of embedded languages in the context of our simulation framework extend far beyond the example given in the previous subsection. We can for example easily extend our language definition with constructs for parallel composition, and introduce blocking mutable references for communication

between threads. An initial interpretation (in the form of a type-class instance) could then be sequential execution, allowing for the simple search of algorithmic bugs in the application. A second instance could then use the Haskell counterparts for parallel composition and block mutable variables to mimic an actual concurrent execution. A third instance could then interact with OS modules inside a SoOSiM simulated system, allowing a developer to observe the interaction between our new language constructs and the operating system.

We said earlier that one of the interpretations of an embedded language description could be a pretty-printed string-representation. Following up on the idea of converting a description to a datatype, we can also interpret our application description as an abstract syntax tree or even a dependency graph. Such a dependency graph could then be used in another instances of our embedded language that facilitates the automatic parallel execution of independent sub-expressions. Again, we can hook up such an instance to our simulator monad, and observe the effects of the distribution of computation and data, as facilitated by our simulated operating system.

IV. Related Work

COTSon [1] is a full system simulator, using an emulator (such as SimNow) for the processor architecture. It allows a developer to execute normal x86-code in a simulated environment. COTSon is far too detailed for our needs, and does not facilitate the easy exploration of a complete operating system.

OMNeT++ [2] is a C++-based discrete event simulator for modelling distributed or parallel system. Compared to SoOSiM, OMNeT++ does not allow the straightforward creation of new modules, meaning the distribution of modules is static. OMNeT++ is thus not meeting our simulation needs to dynamically instantiate new OS modules and application threads.

House [7] is an operating system built in Haskell; it uses a Haskell run-time system allowing direct execution on bare metal. OS modules are executed with the *Hardware* monad, comparable to our *Simulator* monad, allowing direct interaction with real hardware. Consequently, OS modules in House must be implemented in full detail, meaning this approach is not suitable for our exploration needs.

Barrelfish [8] is an OS in which embedded languages are used, amongst other purposes, to define driver interfaces. These embedded languages are also implemented in Haskell. The approach used in Barrelfish is however to create parsers for their embedded languages so that they may have a *nicer* syntax, inducing an additional implementation burden.

V. Conclusions

Although the SoOSiM simulator is still considered work in progress, it has already allowed us to formalize the interactions between the different OS modules devised within the S(o)OS [3] project. We believe that this is the strength of our simulator's approach: the quick exploration and formalization of system concepts. Fast exploration is achieved by the highly abstracted view of SoOSiM on the hardware / system. However, having to actually program all our OS modules forces us to formalize the interactions within the system; exposing any potential flaw not discovered by an informal (text-based) description of the operating system.

By using embedded languages to program applications that run in our simulated environment, we attain complete control of its execution. By using specific interpretations of our embedded language, we can easily observe specific parts (such as memory access) of an application's execution. Using Haskell functions to specify our embedded language constructs saves us from a high implementation burden usually associated with the creation of the tools / compilers for programming languages.

VI. Future Work

At the moment, the simulation core of SoOSiM is single-threaded. We expect that as we move to the simulation of systems with 10's to 100's of computing nodes, that the single threaded approach can become a performance bottleneck. Although individual components are susceptible for parallel execution, the communication between components is problematically non-deterministic. We plan to use Haskell's implementation of software transactional memory (STM) to safely deal with the non-deterministic communication and still achieve parallel execution.

We will additionally explore the use of embedded languages, in the domain of operating system and programming language design, further. Within the context of the S(o)OS project, we intend to add both explicit parallel composition to our embedded language definition, and implicit parallel interpretation of data-independent sub-expressions. We also intend to implement software transactional memory constructs, and investigate their interaction with the operating system.

Acknowledgements

The authors would like to thank Ivan Perez for the design and implementation of the SoOSiM GUI.

References

[1] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "COTSon: Infrastructure for full system simulation," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 1, pp. 52–61, Jan. 2009.

[2] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," in *Proceedings of Simutools '08*, ICST, Brussels, Belgium, 2008, pp. 1–10.

[3] L. Schubert, A. Kipp, B. Koller, and S. Wesner, "Service-oriented operating systems: future workspaces," *Wireless Communications, IEEE*, vol. 16, no. 3, pp. 42–50, june 2009.

[4] B. O'Sullivan, J. Goerzen, and D. Stewart, *Real World Haskell*, 1st ed. O'Reilly Media, Inc., 2008.

[5] Haskell Wiki. (2012, May) Embedded domain specific language. [Online]. Available: http://www.haskell.org/haskellwiki/Embedded_domain_specific_language

[6] J. Carette, O. Kiselyov, and C.-c. Shan, "Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages," *J. Funct. Program.*, vol. 19, no. 5, pp. 509–543, Sep. 2009.

[7] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach, "A Principled Approach to Operating System Construction in Haskell," in *Proceedings of ICFP '05*. New York, NY, USA: ACM, 2005, pp. 116–128.

[8] P.-E. Dagand, A. Baumann, and T. Roscoe, "Filet-o-Fish: practical and dependable domain-specific languages for OS development," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 35–39, Jan. 2010.