Sang Lyul Min
Robert Pettit
Peter Puschner
Theo Ungerer (Eds.)

LNCS 6399

Software Technologies for Embedded and Ubiquitous Systems

8th IFIP WG 10.2 International Workshop, SEUS 2010
Waidhofen/Ybbs, Austria, October 2010
Proceedings

ifip

Lecture Notes in Computer Science 6399

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Sang Lyul Min Robert Pettit
Peter Puschner Theo Ungerer (Eds.)

Software Technologies for Embedded and Ubiquitous Systems

8th IFIP WG 10.2 International Workshop, SEUS 2010
Waidhofen/Ybbs, Austria, October 13-15, 2010
Proceedings

Springer

Volume Editors

Sang Lyul Min
Seoul National University, School of Computer Science and Engineering
599 Kwanak-ro, Gwanak-gu, Seoul 151-742, Korea
E-mail: symin@snu.ac.kr

Robert Pettit
The Aerospace Corporation, Software Systems Engineering Department
15049 Conference Center Drive, Chantilly, CH3/320, VA 20151, USA
E-mail: rpettit@gmu.edu

Peter Puschner
Vienna University of Technology, Institute of Computer Engineering
Treitlstrasse 3, 1040 Vienna, Austria
E-mail: peter@vmars.tuwien.ac.at

Theo Ungerer
University of Augsburg, Systems and Networking
Universitätsstr. 6a, 86135 Augsburg, Germany
E-mail: ungerer@informatik.uni-augsburg.de

Library of Congress Control Number: 2010935603

CR Subject Classification (1998): D.2, C.2, H.4, D.3, K.6, D.1

LNCS Sublibrary: SL 3 – Information Systems and Application, incl. Internet/Web and HCI

ISSN 0302-9743
ISBN-10 3-642-16255-X Springer Berlin Heidelberg New York
ISBN-13 978-3-642-16255-8 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

The 8th IFIP Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS 2010) in Waidhofen/Ybbs, Austria, October 13-15, 2010, succeeded the seven previous workshops in Newport Beach, USA (2009); Capri, Italy (2008); Santorini, Greece (2007); Gyeongju, Korea (2006); Seattle, USA (2005); Vienna, Austria (2004); and Hokodate, Japan (2003); installing SEUS as a successfully established workshop in the field of embedded and ubiquitous systems.

SEUS 2010 continued the tradition of fostering cross-community scientific excellence and establishing strong links between research and industry. SEUS 2010 provided a forum where researchers and practitioners with substantial experiences and serious interests in advancing the state of the art and the state of practice in the field of embedded and ubiquitous computing systems gathered with the goal of fostering new ideas, collaborations, and technologies. The contributions in this volume present advances in integrating the fields of embedded computing and ubiquitous systems.

The call for papers attracted 30 submissions from all around the world. Each submission was assigned to at least four members of the Program Committee for review. The Program Committee decided to accept 21 papers, which were arranged in eight sessions. The accepted papers are from Austria, Denmark, France, Germany, Italy, Japan, Korea, Portugal, Taiwan, UK, and USA. Two keynotes complemented the strong technical program.

We would like to thank all those who contributed to the success of this workshop. First of all, we would like to thank the authors, who sent the papers that made up the essence of this workshop. We are particularly thankful to the General Co-chairs, Peter Puschner and Sang Lyul Min, who organized the entire workshop, to the Program Committee members, and to the additional reviewers, who each contributed their valuable time to review and discuss each of the submitted papers. We would also like to thank the Steering Committee Co-chairs, Peter Puschner, Yunmook Nah, Uwe Brinkschulte, Franz Rammig, Sang Son, and Kane H. Kim for their continual advice and support. Our special thanks go to the members of the Organization Committee for their numerous contributions: Florian Kluge set up the conference software; Julia Schmitt and Michael Roth took over the arduous work of preparing this volume. We especially would like to thank Sibylle Kuster and Maria Ochsenreiter for taking care of the local arrangements and the many aspects of preparing the workshop. Thanks are also to Springer for producing this volume, and last but not least, we would like to thank the IFIP Working Group 10.2 on Embedded Systems, for sponsoring the workshop.

July 2010

Theo Ungerer
Robert Pettit

Organization

General Chairs

Peter Puschner	Vienna University of Technology, Austria
Sang Lyul Min	Seoul National University, Korea

Program Chairs

Theo Ungerer	University of Augsburg, Germany
Robert Pettit	The Aerospace Corporation, USA

Steering Committee

Peter Puschner	Vienna University of Technology, Austria
Yunmook Nah	Dankook University, Korea
Uwe Brinkschulte	University of Frankfurt, Germany
Franz J. Rammig	University of Paderborn, Germany
Sang Son	University of Virginia, Charlottesville, USA
Kane H. Kim	UC Irvine, USA

Program Committee

Luis Almeida	University of Porto, Portugal
Uwe Brinkschulte	University of Frankfurt, Germany
Lynn Choi	Korea University, Korea
Paul Couderc	INRIA, France
Wilfried Elmenreich	University of Klagenfurt, Austria
Gerhard Fohler	TU Kaiserslautern, Germany
Kaori Fujinami	Tokyo University of Agriculture and Technology, Japan
Jan Gustafsson	Malardalen University, Sweden
Leandro Soares Indrusiak	Univerity of York, UK
Vana Kalogeraki	Athens University of Economy and Business, Greece
Soila Kavulya	Carnegie Mellon University, USA
Doohyun Kim	Konkuk University, Korea
Kane H. Kim	UC Irvine, USA
Sunggu Lee	POSTECH, Korea

Steve Meyers	The Aerospace Corporation, USA
Tatsuo Nakajima	Waseda University, Japan
Yukikazu Nakamoto	University of Hyogo and Nagoya University, Japan
Priya Narasimhan	Carnegie Mellon University, USA
Roman Obermaisser	Vienna University of Technology, Austria
Michael Paulitsch	EADS, Germany
Zlatko Petrov	Honeywell International, Czech Republic
Franz J. Rammig	University of Paderborn, Germany
Martin Schoeberl	Technical University of Denmark, Denmark
Eltefaat Shokri	The Aerospace Corporation, USA
Richard Soley	Object Management Group, USA
Sascha Uhrig	University of Augsburg, Germany
Sami Yehia	Thales, France
Rafael Zalman	Infineon, Germany
Wenbing Zhao	Cleveland State University, USA

Subreviewers

Byeonguk Bae	Kisup Hong	Jörg Mische
Faruk Bagci	Rolf Kiefhaber	Peter Puschner
Mike Gerdes	Stefan Metzlaff	Ramon Serna Oliver

Table of Contents

Invited Program

Hardware

Real-Time Systems

Model-Based Design and Model-Checking

Sensor Nets

Error Detection and System Failures

Hard Real-Time

Middleware and Smart Spaces

Function Composition and Task Mapping

Component-Based Design of Embedded Systems

Hermann Kopetz

TU Wien, Austria
hk@vmars.tuwien.ac.at

Abstract. In many engineering disciplines, large systems are built from prefabricated components with known and validated properties. Components are connected via stable, understandable, and standardized interfaces. The system engineer has knowledge about the global properties of the components-as they relate to the system functions-and of the detailed specification of the component interfaces. Knowledge about the internal design and implementation of the components is neither needed, nor available in many cases. A prerequisite for such a constructive approach to system building is that the validated properties of the components are not affected by the system integration. This composability requirement is an important constraint for the selection of a platform for the component-based design of large distributed embedded systems.

Component-based design is a meet-in-the middle design method. On one side the functional and temporal requirements on the components are derived top-down from the desired application functions. On the other side, the functional and temporal capabilities of the components are contained in the specifications of the available components (bottom up). During the design process a proper match between component requirements and component capabilities must be established. If there is no component available that meets the requirements, a new component must be developed.

A prerequisite of any component-based design is a crystal clear component concept that supports the precise specification of the services that are delivered and acquired across the component interfaces. In real-time systems, where the temporal properties of component services are as important as the value properties, the proper notion of a component is a hardware software unit.

In the first part of this presentation the different interfaces of an embedded system component will be introduced. The services of the component are provided by the linking interface (LIF) that must be precisely specified in the domains of value and time. The technology independent interface TII is used to parameterize a component to the given application environment. While a component provider must assure that the component works properly in all configurations that are covered by the parameter space, the component user is primarily interested in the correct operation of the concrete component configuration. The following parts will be concerned with the composition of components and the notion of emerging properties in systems of components.

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, p. 1, 2010.

AUTOSAR Appropriates Functional Safety and Multi-core Exploitation

Bert Böddeker[1] and Rafael Zalman[2]

[1] DENSO AUTOMOTIVE Deutschland GmbH
Eching, Germany
[2] Infineon Technologies AG
Automotive Electronics
Munich, Germany

Abstract. The main subject of this presentation is the connection between AUTOSAR as software standardization initiative and the automotive functional safety domain. The History of AUTOSAR and the functional safety evolution for road vehicles will be presented together with a short overview of the typical implementation of these techniques (E-Gas, X by Wire, etc.).

New emerging multi-core architectures are influencing the automotive related software implementations and have relevance in both software standardization and in the implementation of specific functional safety techniques. The actual status will be presented together with the potential future techniques.

It is discussed how much support AUTOSAR can already provide for practical implementations of functional safety and multi core today followed by an outlook on ongoing research and further standardization needs.

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, p. 2, 2010.

Chip-Size Evaluation of a Multithreaded Processor Enhanced with a PID Controller

Michael Bauer, Mathias Pacher, and Uwe Brinkschulte

Embedded Systems
Goethe-University Frankfurt am Main, Germany
{mbauer,pacher,brinks}@es.cs.uni-frankfurt.de

Abstract. In this paper the additional chip size of a Proportional/Integral/Differential (PID) controller in a multithreaded processor is evaluated. The task of the PID unit is to stabilize a thread's throughput, the instruction per cycle rate (IPC rate). The stabilization of the IPC rate allocated to the main thread increases the efficiency of the processor and also the execution time remaining for other threads. The overhead introduced by the PID controller implementation in the VHDL model of an embedded Java real-time-system is examined.

Keywords: Multithreaded processors, real-time processors, control loops for processors.

1 Introduction

Today it can be observed that electronic devices are getting more and more omnipresent in everybody's life. These embedded systems are small and often have only limited power supply. Processors in embedded systems have to meet different requirements then CPUs in personal computers. The microcontrollers used in such devices have to be fast and energy efficient. Furthermore, the applications running on embedded devices often have real-time requirements.

The most simple solution to handle many threads and to ensure on-time execution for real-time applications would be a fast high-end processor. However, high-end processors consume too much energy for most embedded applications. Furthermore, due to techniques like caches and speculation their real-time performance is hard to predict.

Our solution is to optimize the performance and utilization of a smaller and simpler processor core. We use a PID controller to stabilize the IPC rate for a real time thread. This additional controller reacts adaptively to unpredictable latencies caused by load/store instructions, branches or locks. By reducing the allocated time for a real-time thread to the required rate, the lower priority threads can be executed faster. The smaller consumption of the execution time allows the use of smaller processors.

However, the use of a PID controller increases the energy consumption and needed chip space. In this paper we analyze the additional space needed for the hardware implementation of a PID controller based on VHDL. The logic cells used in a FPGA permit a good estimation of energy and space consumption.

The core we used for our evaluation is an enhancement of the multithreaded Java microcontroller named Komodo [6]. Komodo offers the required capabilities for real

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 3–12, 2010.

time applications and is also multithreaded. Therefore, the mentioned goals of the PID controller can be tested on this processor.

Furthermore, basic settings and requirements have been researched via a PID controller implementation in a software processor simulator [6,5] using different benchmarks. The resulting controller parameters (e.g. width of the controlling window) are now used as a basis for the chip space evaluation.

2 The Komodo Microcontroller

The Komodo microcontroller consists of a processor core attached to typical devices such as a timer/counter, capture/compare, serial and parallel interfaces via an I/O bus [28]. In the following we explain details about the processor core which is kept at a simple hardware level because of its real-time applications.

2.1 The Pipeline

Figure (1) shows the pipeline enhanced by the priority manager and the signal unit. The pipeline consists of the following four stages: instruction fetch (IF), instruction window and decode (ID), operand fetch (OF) and execute, memory and I/O-access (EXE). These stages perform the following tasks as described in [34]:

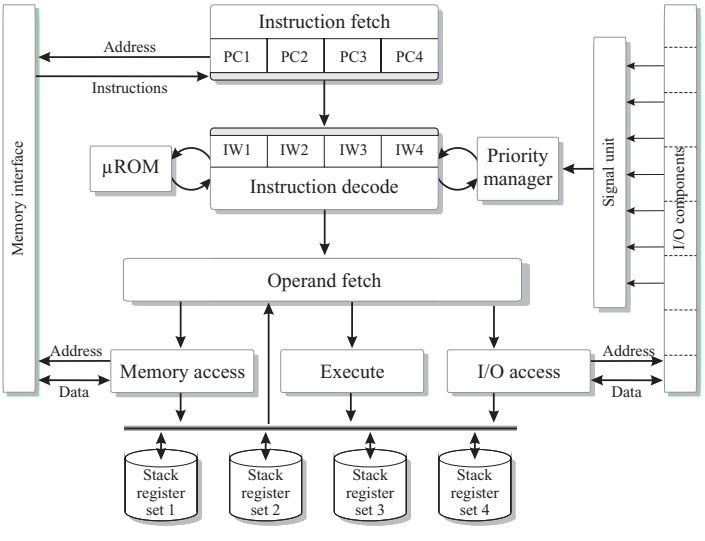


Fig. 1. The pipeline of the Komodo microcontroller

Instruction fetch: The instruction fetch unit tries to fetch a new instruction package from the memory interface in each clock cycle. If there is a branch executed in EX

the internal program counter is set and the instruction package is fetched from the new address.

Instruction window and decode: The decoding of an instruction starts after writing a received instruction package to the correct instruction window. Hereby, the priority manager decides which thread will be decoded by ID in every clock cycle.

Operand fetch: In this pipeline stage operands needed for the actual operation are read from the stack.

Execution, memory and I/O-access: There are three units in the execution stage (ALU, memory and I/O). All instructions but load/store are executed by the ALU. The result is sent to the stack and to OF for forwarding. In case of load/store instructions the memory is addressed by one of the operands. An I/O-access is handled in the same manner as a memory access [29, 34].

2.2 Guaranteed Percentage Scheduling

The priority manager implements several real-time scheduling schemes [9, 18, 34]. For the concern of controlling, only GP scheduling is important. In GP scheduling, the priority manager assigns a requested number of clock cycles to each thread. This assignment is done within a 100 clock cycle period. Figure (2) gives an example of two threads with 20% and 80%, i.e. thread A gets 20 clock cycles and thread B gets 80 clock cycles within the 100 clock cycle interval. Of course, these clock cycles may contain latencies. So thread A might not be able to execute 20 instructions in its 20 clock cycles and this is where our approach starts. By monitoring the real IPC rate, the GP value is adjusted in a closed feedback loop. If there are e.g. 3 latency clock cycles within the 20 clock cycles of thread A, its percentage needs to be adjusted to achieve the desired 20 instructions in the 100 clock cycle interval.

2.3 Measurement of the IPC Rate

In general, the IPC rate of a thread is defined as:

$$IPC = \frac{Number\ of\ instructions\ executed\ within\ time\ slice}{Duration\ of\ time\ slice\ (in\ clock\ cycles)} \tag{1}$$

Since Komodo is single-scalar, each instruction takes one clock cycle, unless there is a latency. If we distinguish between active clock cycles, where an instruction of a thread is really executed, and latency clock cycles, we can refine the definition of the IPC rate for the Komodo microcontroller:

$$IPC = \frac{Number\ of\ active\ clock\ cycles\ within\ time\ slice}{Duration\ of\ time\ slice\ (in\ clock\ cycles)} \tag{2}$$

The number of active clock cycles of a thread is the number of clock cycles executed by a thread without the clock cycles used by latencies. It is obvious that both latencies and locks interfere with the IPC rate. Therefore, the function of the controller is to minimize these interferences with the IPC rate.

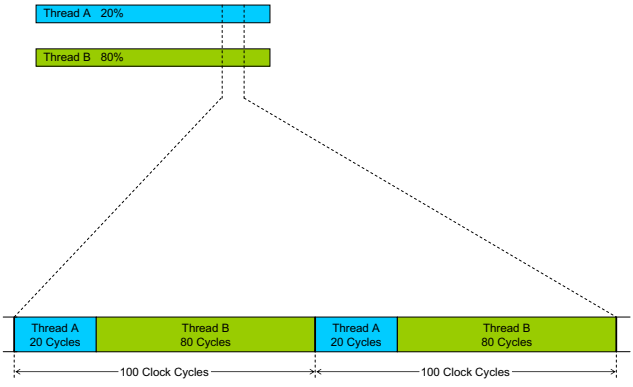


Fig. 2. Example for GP scheduling

We use this formula for different kinds of measures: We compute short- and long-term IPC rates. Short-term IPC rates are measured over a constant time slice of e.g. 400 clock cycles. The cumulative IPC rate is used for the longtime IPC rate. We compute the IPC rate from the beginning of the thread execution up to now, the duration of the time slice is increasing. The short-term IPC rate gives information about variations in the current IPC rate, the cumulative IPC rate shows the overall behavior.

2.4 Use of a PID Controller in a Microprocessor

In system design, the relationship of input and output values of a system is of great importance. Often, this relationship is not exactly known or influenced by unknown disturbances. Here, control theory comes into play. The system input is controlled by observing the output and comparing it with the desired values in order to minimize their difference by varying the input [12,20]. The controller in this closed feedback loop is responsible for generating the control signal y_n from the difference signal x_n. A well known and popular controller is the PID controller. The functional equation for a discrete PID controller is as follows:

$$y_n = K_P * x_n + K_I * \sum_{\nu=1}^{n} x_\nu * \Delta t + K_D * \frac{x_n - x_{n-1}}{\Delta t} \qquad \text{n} = 2, 3, 4... \qquad (3)$$

Hereby x_n is the difference at time n and y_n is the controller signal at time n. K_P, K_I and K_D are the controller constants while Δt is the duration between the measurement of x_n and x_{n+1}.

The PID controller works in the processor as is shown in Figure 3. The current IPC rate is computed by a measurement unit. The PID controller compares this value with the reference IPC value by computing their difference. Then it uses the PID algorithm from equation (3) to compute the new GP value for a thread to be controlled.

As shown in the related work section 4 the control approach as described above works really well. Therefore, it is important to evaluate the size of the controller relative

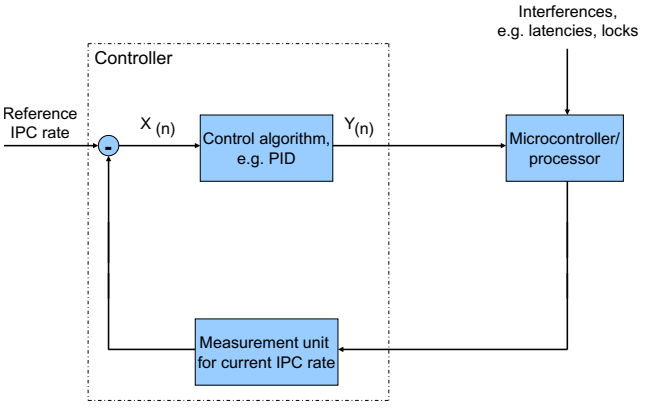


Fig. 3. Use of a controller to stabilize the throughput

to the size of the microprocessor to estimate the practicability of the use of such a controller in a FPGA or even an ASIC implementation; this work is described in the next section.

3 Evaluation of the Processor Size

3.1 Preliminary Work for the PID Controller Implementation in VHDL

Previous work on a PID controller for IPC rate stabilization was already carried out via a software processor simulator, based on the Komodo architecture [6,5]. The simulator was written in Java code. This allowed easy and quick changes to check different configurations of the PID controller. As a result, tests values for the following essential parameters for the succeeding VHDL implementation have been determined.

- The values of K_P, K_I and K_D depend on the current running applications and the kind of appearing latencies. It was not part of this work to find proper strategies to select these constants. So values of prior simulation runs were taken for the PID controller's VHDL implementation.
- The PID controller has to generate the new signal every 100 cycles. The current IPC rate is also sampled every 100 cycles.
- For stable controller operation, the IPC rate should be calculated as an average value over a longer period of time. However, if this period gets too long, the controller will react lazy. Simulation results have shown that 1000 clock cycles (= 10 controller cycles) are a very good value as a length of the controlling window.

These boundaries define the structure and required space for the VHDL design.

Another approach to the previous work on the simulator was the reuse of the GP scheduler. The PID controller signal y_n directs the IPC rate via this scheduler. Therefore, the period which the PID controller needs to generate a new signal is the same as the GP scheduling window. So, the PID controller simply regulates the allocated GP

value within a range of 0 to 100 to reach the desired IPC rate. The former input of the GP scheduler is used as a set value for the PID controller and the signal y_n forms the new input of the scheduler.

3.2 Implementation of the PID Controller in VHDL

The PID controller has been integrated in an enhanced Komodo IP core. The GP scheduler, which was already implemented in the IP core, has a scheduling window of 100 cycles. This dictates the rate of the PID controller. The upcoming operations of one PID controlling window of 100 cycles are:

- Calculating the average difference value x_n for the last 1000 cycles. This period corresponds to the last 10 GP scheduling windows (see figure 4). The overlap between the current and the prior PID controlling windows consist of the previous 9 GP scheduling windows. Regarding this overlap the new value of x_n can be calculated by modifying the average difference value of the prior PID controlling windows x_{n-1}. Therefore the difference value of 10th oldest GP scheduling window has to be subtracted and the difference value of the current GP scheduling window has to be added to x_{n-1}.

  So, the history containing the differences between actual and set values of the last 10 GP scheduling windows has to be stored.

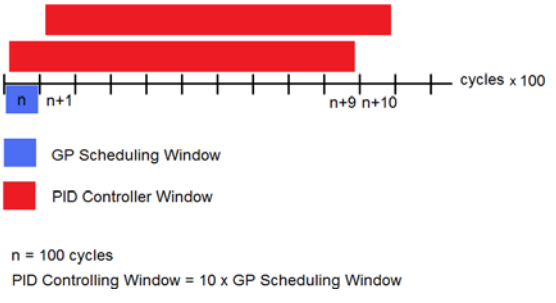


Fig. 4. Relation of PID Controlling Window and GP Scheduling Window

- To calculate the P component of the PID controller equation, $K_P * x_n$, one multiplication has to be carried out.
- To calculate the D component of the PID controller equation, $K_D * \frac{x_n - x_{n-1}}{\Delta t}$, one multiplication of the constant value $\frac{K_D}{\Delta t}$ by the result of $x_n - x_{n-1}$ has to be carries out. Regarding the way x_n was calculated, the difference of $x_n - x_{n-1}$ can be calculated by simply subtracting the difference value of the 10th oldest GP scheduling window from the value of the current GP scheduling window.
- To calculate the I component of the PID controller equation, $K_I * \sum_{\nu=1}^{n} x_\nu * \Delta t$, one multiplication of the constant value of $(K_I * \Delta t)$ by the result of $\sum_{\nu=1}^{n} x_\nu$ has to be carried out. This sum is calculated recursively by adding the current x_n to the sum $\sum_{\nu=1}^{n-1} x_\nu$ calculated in the PID controlling window before.

The implementation of the PID controller on an Altera Cyclon II FPGA board showed the following result: The required "dedicated logic register" increased by 3 percent and the number of "combinational functions" by 1.4 percent. The number of total logic elements increased by 2 percent overall.

Table 1. Results of the PID Controller Space Evaluation

IP Core:	without PID Controller	with PID Controller
Dedicated Logic Registers	4688	4829
Total Combinational Functions	12696	12873
Total Logic Elements	12895	13147
Space used on the Cyclon II FPGA	39%	40%

This is very promising result gives reason to spend more research on space requirements for various controller and thread settings.

4 Related Work

Hardware multithreading has made its way from pure research to a state-of-the-art processor technique.

Early examples for multithreaded processors are Cray MTA (a high performance multi-threaded VLIW processor [2]) and Cray MTA2 (supports up to 128 RISC-like hardware threads [11]). The MIT Sparcle processor is a classical example based on the SPARC RISC processor supporting up to 4 threads and context switching on long memory latencies [1]. The MSparc processor supporting up to 4 threads [24] and real-time properties [19,23] can be cited in this context as well.

In the embedded field, multithreading has been introduced by the two-threaded Infineon TriCore II [27] or the MIPS32-34K [25] with up to nine thread contexts. Research-oriented small processors like the Komodo microcontroller [17,16] with a scalar four-threaded Java processor core and the CAR-SoC [15,26] with a SMT processor core investigate the use of multithreading for real-time embedded applications. Real-time scheduling for SMT processors is also researched in [10]. Here, a resource allocator is used to control shared resources (e.g. registers, issue bandwidth, buffers, etc.) of threads to control the throughput. Another multithreaded processor featuring real-time scheduling is the Jamuth IP core [35], which is an enhancement of the Komodo microcontroller.

Furthermore, multithreading is often combined with *multi-core architectures*. Multi-core processors combine several processor cores on a single chip. In many cases the core is a multithreaded core.

Commercial examples for multi-core multithreaded processors are e.g the Sun Niagara (Sun UltraSPARC T1 [31]) with up to eight four-threaded processor cores or the IBM Power series. IBM Power5 and Power6 [13,14] contain 2 SMT processor cores with shared secondary level cache supporting dynamic resource scheduling for threads [3]. Furthermore, multi-chip modules containing several processor and third level cache chips are available. The upcoming Power7 architecture is announced to contain up to

eight processor cores [30]. Intel processors like the Core 2 Duo, Core 2 Quad or their successors Core i3, i5, i7 and i9 are also equipped with multiple cores supporting multithreading. The Intel variant of multithreading is called hyperthreading [21].

Research multi-core processors are e.g. MIT RAW with 16 MIPS-style cores and a scalable instruction set [32] or AsAP (*As*ynchronous *A*rray of simple *P*rocessors) containing 36 (AsAP 1 [36]) or 167 (AsAP 2 [33]) cores dedicated to signal processing. Researching real-time support on such processors, the MERASA project has to be cited [22]. This project combines execution time analysis like worst case execution time or longest execution time with multi-core and SMT microarchitectures for hard real-time systems.

Our approach is completely different from the methods mentioned above where no control theory is used. Some of the approaches mentioned above (e.g. [10,3,13,14]) support dynamic adaptation of thread parameters and resources. However, according to our best knowledge our approach is the only one using control theory as a basis. The improvements achieved by this approach have been shown in several publications [6,5,8,4]. As a major advantage, mathematical models of control theory can be used to guarantee real-time boundaries. Furthermore, we tested the parameters of the PID controller's algorithm on a simulator of the Komodo microcontroller [7].

5 Conclusion and Further Work

Controlling the throughput of a thread by a PID controller has already been successfully tested in the preceding work via a software simulator. In this paper the focus is on the additional required space for the PID controller. Therefore the parameters determined by the simulator were used to build and configure a VHDL model of the controller. As a result of the evaluation, only 2 percent of additional overhead has been created.

An interesting future approach will be to test the configurations in the VHDL model under various thread settings. Depending on the results obtained, optimizations of the VHDL code could reduce the required space. Many parameters like the execution period of the PID controller, the period of calculating the average IPC rate and the history size to calculate the integral part of the PID controller influence the size of the controller.

Also the combination of different types of controlling strategies for threads is of interest with respect to accuracy, speed and real time constraints. The use of a combination of controlling strategies regarding different thread scenarios is an interesting future approach.

References

1. Agarwal, A., Kubiatowicz, J., Kranz, D., Lim, B.H., Yeoung, D., D'Souza, G., Parkin, M.: Sparcle: An Evolutionary Processor Design for Large-scale Multiprocessors. IEEE Micro 13(3), 48–61 (1993)
2. Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A., Smith, B.J.: The Tera Computer System. In: 4th International Conference on Supercomputing, Amsterdam, The Netherlands (1990)

3. Borkenhagen, J.M., Eickemeyer, R.J., Kalla, R.N., Kunkel, S.R.: A Multithreaded PowerPC Processor for Commercial Servers. IBM Journal Research and Development 44, 885–898 (2000)
4. Brinkschulte, U., Lohn, D., Pacher, M.: Towards a statistical model of a microprocessor's throughput by analyzing pipeline stalls. In: Lee, S., Narasimhan, P. (eds.) SEUS 2009. LNCS, vol. 5860, pp. 82–90. Springer, Heidelberg (2009)
5. Brinkschulte, U., Pacher, M.: Implementing control algorithms within a multithreaded java microcontroller. In: Beigl, M., Lukowicz, P. (eds.) ARCS 2005. LNCS, vol. 3432, pp. 33–49. Springer, Heidelberg (2005)
6. Brinkschulte, U., Pacher, M.: Improving the real-time behaviour of a multithreaded java microcontroller by control theory and model based latency prediction. In: WORDS, pp. 82–96 (2005)
7. Brinkschulte, U., Pacher, M.: Improving the Real-time Behaviour of a Multithreaded Java Microcontroller by Control Theory and Model Based Latency Prediction. In: WORDS 2005, Tenth IEEE International Workshop on Object-oriented Real-time Dependable Systems, Sedona, Arizona, USA (2005)
8. Brinkschulte, U., Pacher, M.: A control theory approach to improve the real-time capability of multi-threaded microprocessors. In: ISORC, pp. 399–404 (2008)
9. Brinkschulte, U., Ungerer, T.: Mikrocontroller und Mikroprozessoren, vol. 2. Springer, Heidelberg (2007)
10. Cazorla, F., Fernandez, E., Knijnenburg, P., Ramirez, A., Sakellariou, R., Valero, M.: Architectural Support for Real-Time Task Scheduling in SMT-Processors. In: CASES'05, San Francisco, California, USA (September 2004)
11. Cray: Ceay MTA-2 System (2001), http://www.netlib.org/utk/papers/advanced-computers/mta-2.html
12. Dorf, R., Bishop, R.: Modern Control Systems. Addison-Wesley, Reading (2002)
13. IBM: Power6 Specifications (2007), http://www-03.ibm.com/press/us/en/attachment/21546.wss?fileId=ATTACH_FILE1&fileName=POWER6%20Specs.pdf
14. IBM: IBM POWER systems (2009), http://www-03.ibm.com/systems/power/
15. Kluge, F., Mische, J., Uhrig, S., Ungerer, T.: CAR-SoC - Towards and Autonomic SoC Node. In: Second International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2006), L'Aquila, Italy (July 2006)
16. Kreuzinger, J., Brinkschulte, U., Pfeffer, M., Uhrig, S., Ungerer, T.: Real-time Event-handling and Scheduling on a Multithreaded Java Microcontroller. Microprocessors and Microsystems Journal 27(1), 19–31 (2003)
17. Kreuzinger, J., Pfeffer, M., Ungerer, T., Brinkschulte, U., Krakowski, C.: The Komodo Project: Real-Time Java Based on a Multithreaded Java Microcontroller. In: PDPTA 2000 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, USA (2000)
18. Kreuzinger, J.: Echtzeitfähige Ereignisbehandlung mit Hilfe eines mehrfädigen Java-Mikrocontrollers. Ph.D. thesis, Logos Verlag Berlin (2001)
19. Lüth, K., Metzner, A., Piekenkamp, T., Risu, J.: The EVENTS Approach to Rapid Prototyping for Embedded Control System. In: Workshop Zielarchitekturen Eingebetteter Systeme, Rostock, Germany, pp. 45–54 (1997)
20. Lutz, H., Wendt, W.: Taschenbuch der Regelungstechnik. Verlag Harri Deutsch (2002)
21. Marr, D., Binns, F., Hill, D., Hinton, G., Koufaty, D., Miller, J., Upton, M.: Hyper-Threading Technology Architecture and Microarchitecture: A Hypertext History. Intel. Technology Journal 6 (2002)
22. MERASA: Multi-Core Execution of Hard Real-Time Applications Supporting Analysability (2009), http://ginkgo.informatik.uni-augsburg.de/merasa-web/

23. Metzner, A., Niehaus, J.: MSparc: Multithreading in Real-time Architectures. J. Universal Comput. Sci. 6(10), 1034–1051 (2000)
24. Mikschl, A., Damm, W.: MSparc: a Multithreaded Sparc. In: Fraigniaud, P., Mignotte, A., Bougé, L., Robert, Y. (eds.) Euro-Par 1996. LNCS, vol. 1123, pp. 461–469. Springer, Heidelberg (1996)
25. MIPS Technologies, Inc: The MIPS32-34K Core Family: Powering Next-Generation Embedded SoCs. Research Report pp. 1034–1051 (September 2005)
26. Nickschas, M., Brinkschulte, U.: Guiding Organic Management in a Service-Oriented Real-Time Middleware Architecture. In: Brinkschulte, U., Givargis, T., Russo, S. (eds.) SEUS 2008. LNCS, vol. 5287, pp. 90–101. Springer, Heidelberg (2008)
27. Norden, E.: A Multithreaded RISC/DSP Processor with High Speed Interconnect. Hot Chips 15 (2003)
28. Pfeffer, M., Ungerer, T., Uhrig, S., Brinkschulte, U.: Connecting peripherial interfaces to a multi-threaded java microcontroller. In: Workshop on Java in Embedded Systems, ARCS 2002 (2002)
29. Pfeffer, M.: Ein echtzeitfähiges Javasystem für einen mehrfädigen Java-Mikrocontroller. Ph.D. thesis, Logos Verlag Berlin (2004)
30. Stokes, J.: IBM's 8-core POWER7: twice the muscle, half the transistors. Ars Technica (September 2009), http://arstechnica.com/hardware/news/2009/09/ibms-8-core-power7-twice-the-muscle-half-the-transistors.ars
31. Sun: UltraSPARC T1 Processor (2005), http://www.sun.com/processors/UltraSPARC-T1/
32. Taylor, M., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffmann, H., Johnson, P., Lee, J., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., Agarwal, A.: The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. IEEE Micro (March/April 2002)
33. Truong, D.N., Cheng, W.H., Mohsenin, T., Yu, Z., Jacobson, A.T., Landge, G., Meeuwsen, M.J., Tran, A.T., Xiao, Z., Work, E.W., Webb, J.W., Mejia, P., Baas, B.M.: A 167-processor computational platform in 65 nm cmos. IEEE Journal of Solid-State Circuits (JSSC) 44(4), 1130–1144 (2009)
34. Uhrig, S., Liemke, C., Pfeffer, M., Becker, J., Brinkschulte, U., Ungerer, T.: Implementing real-time scheduling within a multithreaded java microcontroller. In: 6th Workshop on Multithreaded Execution, Architecture, and Compilation MTEAC-6, in Conjunction with 35th International Symposium on Microarchitecture MICRO-35, Istanbul (November 2002)
35. Uhrig, S., Wiese, J.: jamuth: an ip processor core for embedded java real-time systems. In: JTRES, pp. 230–237 (2007)
36. Yu, Z., Meeuwsen, M., Apperson, R., Sattari, O., Lai, M., Webb, J., Work, E., Mohsenin, T., Singh, M., Baas, B.M.: An asynchronous array of simple processors for dsp applications. In: IEEE International Solid-State Circuits Conference (ISSCC '06), pp. 428–429 (February 2006)

Crash Recovery in FAST FTL*

Sungup Moon[1], Sang-Phil Lim[1], Dong-Joo Park[2], and Sang-Won Lee[1]

[1] Sungkyunkwan University, Cheoncheon-Dong, Jangan-Gu, Suwon, Gyeonggi-Do 440-746, Korea
[2] Soongsil University, Sangdo-Dong, Dongjak-Gu, Seoul 156-743, Korea
{sungup,lsfeel0204}@skku.edu, djpark@ssu.ac.kr, swlee@skku.edu

Abstract. NAND flash memory is one of the non-volatile memories and has been replacing hard disk in various storage markets from mobile devices, PC/Laptop computers, even to enterprise servers. However, flash memory does not allow in-place-update, and thus a block should be erased before overwriting the existing data in it. In order to overcome the performance problem from this intrinsic deficiency, flash storage devices are equipped with the software module, called FTL (Flash Translation Layer). Meanwhile, flash storage devices are subject to failure and thus should be able to recover metadata (including address mapping information) as well as data from the crash. In general, the FTL layer is responsible for the crash recovery. In this paper, we propose a novel crash recovery scheme for FAST, a hybrid address mapping FTL. It writes periodically newly generated address mapping information in a log structured way, but it exploits the characteristics of FAST FTL that the log blocks in a log area are used in a round-robin way, thus providing two advantages over the existing FTL recovery schemes. One is the low overhead in performing logging during normal operations in FTL. The other is the fast recovery time.

1 Introduction

For the past decade, we have witnessed that flash memory is deployed as an alternative data storage for mobile devices, laptops, and even enterprise server applications. Mainly because of the erase-before-update characteristics in flash memory, every flash memory storage comes with a core software module, flash translation layer(FTL). Since FTL can critically determine the performance of a flash device, numerous FTLs have been proposed. And taking into account that the capacity of flash memory devices drastically increases, the efficiency of FTLs becomes more and more important. According to address mapping, the existing FTLs can be categorized largely into block mapping, page mapping, and hybrid mapping.

However, most existing works on FTL has been focused on issues such as performance, wear leveling, and garbage collections. In contrast, despite of its

* This research was supported by MKE, Korea under ITRC NIPA-2010-(C1090-1021-0008) and Seoul Metropolitan Government under 'Seoul R&BD Program (PA090903).'

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 13–22, 2010.

practical importance, the recovery from power-off failure in FTLs has not been paid much attention to. As far as we know, there is no work which deals with the crash recovery in FTLs comprehensively. When a page write is requested, a new address mapping information, in addition to the data page itself, should also be persistently propagated to flash for power-off recovery, either in page mapping or hybrid mapping. Thus, under a naive solution like this, a page write request in flash memory would at least need two physical flash writes, which could degrade the FTL performance. Of course, both the algorithmic complexity and its run-time overhead in FTL may considerably depend on the address mapping being used. For example, in case of block mapping, we have to write the changed address mapping information, only when the block mapping changes, not for each page write. Thus, the recovery solution in block mapping would be very simple and does not incur much run time overhead. In contrast, for every data page write in page mapping FTL, a new page mapping entry should be written in flash memory. In order to alleviate this overhead, we can use a checkpoint approach for flushing new page-level mapping entries periodically in a log structured manner. But, even this approach has the garbage collection overhead for the old page-level mapping entries.

In this paper, we propose a novel crash recovery mechanism for FAST, a hybrid address mapping FTL. In particular, it writes the new mapping information periodically in a log structured way, but it exploits the characteristics of FAST FTL that the log blocks in a log area are used in a round-robin way, thus providing two advantages over other FTL recovery schemes. One is the low overhead in writing the new mapping metadata during the normal operation in FTL. In fact, our recovery scheme will require a nominal size of mapping metadata to be written per data page write. In terms of additional write overhead for recovery, our scheme could outperform the existing approach by up to an order of magnitude. The other advantage is the fast recovery time. With our scheme, the recovery phase can finish as early as possible, and then the FAST FTL proceed in a normal mode. In addition, this paper deals with the duality of flash write overhead between original data page and the metadata for recovery, and we argue that, for comparing FTLs in terms of performance, we should take into account the overhead in maintaining the metadata persistently for recovery.

This paper is organized as follows. Section 2 describes the related work of flash memory and the address mapping table management, and then we explain our recovery method in section 3. In Section 4, we evaluate the management cost arithmetically. Finally, Section 5 concludes this paper.

2 Background and Related Work

2.1 Background: Flash Memory, FTL, and Address Mapping

With flash memory, no data page can be updated in place without erasing a block of flash memory containing the page. This characteristic of flash memory is called “erase-before-write” limitation. In order to alleviate the erase-before-write problem in flash memory, most flash memory storage devices are equipped

with a software or firmware layer called Flash Translation Layer (FTL). An FTL makes a flash memory storage device look like a hard disk drive to the upper layers. One key role of an FTL is to redirect each logical page write from the host to a clean physical page, and to remap the logical page address from the old physical page to the new physical page. In addition to this address mapping, an FTL is also responsible for data consistency and uniform wear-leveling.

By the granularity of address mapping, FTLs can be largely classified into three types: page mapping FTLs [2], block mapping FTLs [1], and hybrid mapping FTLs including BAST[4] and FAST [5]. In a block mapping FTL, address mapping is done coarsely at the level of block. When the data in a block is to be overwritten, FTL assigns a new empty block for the block. Although this inflexible mapping scheme often incurs significant overhead for page writes, we need to change the block mapping information and store the new mapping information persistently only when a new physical block is assigned to a logical block.

In a page mapping FTL, on the other hand, address mapping is finer-grained at the level of pages, which are much smaller than blocks. The process of address mapping in a page mapping FTL is more flexible, because a logical page can be mapped to any free physical page. When an update is requested for a logical page, the FTL stores the data in a clean page, and updates the mapping information for the page. When it runs out of clean pages, the FTL initiates a block reclamation in order to create a clean block. The block reclamation usually involves erasing an old block and relocating valid pages. However, a page mapping FTL requires a much larger memory space to store a mapping table. Because it has a large address mapping table. In order to guarantee the consistent mapping information against the power-off failure, a page mapping FTL basically stores the whole page mapping table for every page write and this overhead is not trivial if we consider the large page mapping table. In a page mapping table, another way to recover the consistent page mapping information from failure is to store the logical page address in the spare area of each physical page, but this approach will suffer from reconstructing the mapping information by scanning all the pages.

Hybrid mapping FTLs have been proposed to overcome the limitations of both page and block mapping FTLs, namely, the large memory requirement of page mapping and the inflexibility of block mapping. In FAST FTL [4], one of the popular hybrid mapping FTLs, flash memory blocks are logically partitioned into a data area and a log area. Blocks in a data area are managed by block-level mapping, while blocks in a log area are done by page-level mapping. If there is a data write request, the FAST scheme will write the data on clean pages at the end of the log area which are managed by the page level address table. Thus, this hybrid FTL scheme retains higher space utilization than the block level mapping scheme and smaller table size than the page level mapping scheme. But, as in the page mapping FTL scheme, every page write will change the page mapping for the log area so that FAST also has the overhead to store, in addition to the data page itself, the new page address mapping entry persistently for each write. However, as will be described later, there is a unique opportunity in FAST FTL for maintaining the page mapping information consistently with lower overhead.

2.2 Crash Recovery in FTLs

Many recovery mechanisms have been proposed to maintain the mapping information, but those mechanisms should work with the upper software layer like the file system. In the decoupled FTL, filesystem layer cannot access the FTL internal data structure like the page mapping table. On the other hand, except for some paper like LTFTL[6] and PORCE[3], the FTL-specific recovery mechanism has not been researched for the SSD's decoupled FTL.

LTFTL [6] is a page mapping FTL with internal recovery scheme. LTFTL keeps the changed address log at the RAM and checkpoints the logs by a unit page. In this scheme, LTFTL can restore not only the latest address table but also other previous address tables through few page accesses. But LTFTL has to merge the logs with the large address table and write the newly whole address table at the mapping block, if the logs become over the log threshold. This transaction affects the normal I/O request performance because of their large-size address table.

PORCE [3] only focused on the recovery scheme after the power-failure. PORCE divided the power-failure problem into two situations; the normal write operation consistency and the reclaiming operation consistency. Especially in the reclaiming operation, PORCE writes the reclaiming-start-log before the reclaiming operation and the reclaiming-commit-log after the all reclaiming operation at the transaction log area. But FAST FTL always performs their merge operation with many associated data block, and each associated block state will be changed every merging step. Thus reclaiming operation of FAST FTL should be traceable and robust against the repetitive recovery-failure, but PORCE did not mentioned about this.

3 Crash Recovery in FAST FTL

3.1 Overview of FAST FTL

FAST FTL is one of the hybrid FTL schemes which was originally designed to improve the performance of small random write. In FAST FTL, flash memory blocks are logically partitioned to a data area and a log area. Blocks in a data area are managed by block-level mapping, while blocks in a log area are done by page-level mapping. Since no single log block is tied up with any particular data block due to its full associativity, any page update can be done to any clean page in any one of the log blocks. This improves the block utilization significantly and avoids the log block trashing problem as well.

By the way, every write in a log area will change the page-level mapping entry for the logical page being written, and thus each write in a log area requires at least *one additional flash write* for storing the page-level mapping information persistently, thus guaranteeing the consistency of the page-level mapping information against the power-off failure. This additional flash write for storing mapping information will degrade the performance by half. Thus, a more clever

solution for crash recovery in FAST is necessary, and we propose a checkpoint-based recovery scheme for FAST FTL.

One concern which makes the recovery in FAST FTL more complex is the full merge operation in FTL. When the log area is full, a log block is chosen as a victim for reclamation and the victim log block may contain valid pages belonging to many different data blocks. In FAST, log blocks are managed in a FIFO queue, and thus the oldest log block is chosen as a victim block. Those valid pages should then be merged with their corresponding data blocks. This type of merge operation is called a full merge, and is usually very expensive as it involves performing merge operations as many as valid pages in the victim log block. Please see [5] for details. Besides its performance overhead, we should note that one victim log block in FAST may incur many merges, and a crash can occur at any point while handling them. For each block being merged, we need to change its block mapping information and save it in flash memory, too. That is, the reclamation of a victim block can change the block mapping information of several blocks being merged, and the crash between the multiple merges makes it hard to achieve the atomic propagation of the block-level mapping changes from multiple full merges. We will revisit this issue later.

Before closing this subsection, let us explain two assumptions we make in this paper. First, both the free block list (i.e. the list of free flash blocks) and the associated-data-block list (i.e. the list of data blocks to be merged when a victim log block is reclaimed) are maintained as sorted lists. Second, the victim log block should be returned to the free block list, instead of being reused as the new log block immediately after reclamation.

3.2 Revised FAST FTL Architecture

In order to support crash recovery functionality to FAST FTL, we need to revise the previous FAST architecture a little, which is described in this subsection.

Address-Mapping Tables. FAST FTL maintains two address-mapping tables, a block mapping table for data blocks and a page mapping table for log blocks. For merge correctness, two new columns, MT(Merge Timestamp) and VT(Victim Timestamp) are added to the two mapping tables, respectively, which will be handled in details in Section 3.3.

Logging Areas on Flash Memory. For the fast recovery from system failures, address mapping information in main memory has to be persistently stored somewhere in flash memory. This is why two logging areas in the front of flash memory exist in Fig. 1. The block mapping logging area stores two classes of block mapping information for the data and log blocks. First, whenever the block mapping table for data blocks is updated due to page writes or merge operations, the whole table is stored in the block mapping logging area. This logging cost is not so expensive since such a update does not happen frequently. Second, the recovery module in FAST FTL is required to know what physical block each

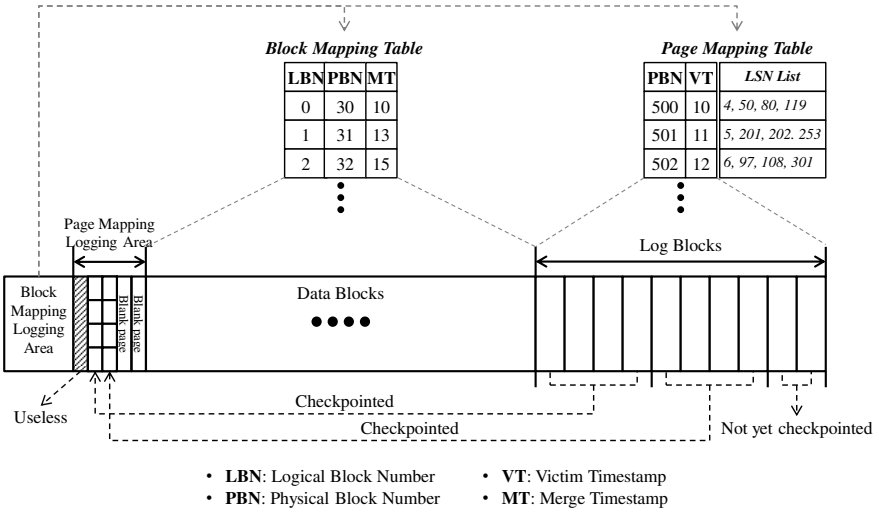


Fig. 1. Revised FAST FTL Architecture

of the log blocks are mapped to before the crash, so this information should be stored onto flash memory whenever address mapping between log blocks and physical blocks changes. Of course, the frequency of these changes is not so high, and thus the cost is low.

Next, let us explain how to use the page mapping logging area. The rest of address mapping information, that is, the information of what logical pages each of the log blocks buffers has to be saved up somewhere of flash memory. In fact, the size of this information is very large, therefore it is too expensive to store this information wholly whenever the page mapping table is updated. This is why we devise a new logging technique to store this-like page mapping information efficiently, which is described in details in Section 3.3.

3.3 Logging in FAST FTL

We described why the page mapping logging area on flash memory is needed in Section 3.2. Here, we explain how the page mapping information is stored and managed in this logging area. It is not efficient to store the whole data of the page mapping table for log blocks every update. Instead, we employ the checkpoint technique, which will acts as follows. As page updates from the file system are done onto the log blocks, the page mapping table for log blocks will grow. The moment the first N log blocks are exhausted, the first N entries of the page mapping table are output into the page mapping logging area. Here N determines the number of these entries whose total size approximates to the page size. In this way, the next N entries will be written into the page mapping logging area. This log containing N entries is called FASTcheck. If later the log block buffer becomes full, the first victim log block selection has to be performed.

Accordingly, the entry information in the first FASTcheck corresponding to this victim log block becomes invalid and afterwards, if the first N victim selections are done, the first FASTcheck in the page mapping logging area becomes invalid completely. If the logging area becomes full by writing a lot of FASTchecks, a new free block is allocated for the new page mapping logging block and then all valid FASTchecks of the old logging block are copied into the new logging block. The old logging block is returned to e.g., the garbage collector.

Since checkpoints are performed per N log blocks, that is, N entries of the page mapping table, the entire page mapping table cannot be persistently stored in flash memory. If a system failure occurs at the time when the last N entries of the page mapping table is not yet accumulated after the last checkpoint is done, this uncheckpointed entries has to be recovered by scanning the corresponding log blocks and extracting their metadata. We will handle this-like recovery issue in the next subsection.

3.4 Recovery in FAST FTL

Prior to the description of crash recovery in FAST, we first assume that a crash never happens while FAST performs a full merge operation after a victim log block is selected. It is noted that a full merge operation consists of multiple individual merge operations. If a crash occurs during full merge, consistency between the block mapping table and the page mapping table cannot be guaranteed and therefore we have to make more efforts to solve this problem. So we plan to handle this complicated issue in the future work. The crash recovery is processed as follows. First, we recover the block mapping table for data blocks from the block mapping logging area and then block mapping information belonging to the page mapping table for log blocks. It is noted that the page mapping table consists of block and page mapping information. Next, we recover the rest of the page mapping table, that is, page mapping information by reading only valid FASTchecks from the page mapping logging area. As mentioned in Section 3.3, the uncheckpointed entries of the page mapping table do not exist in the page mapping logging area. Accordingly, we have to scan log blocks corresponding to these entries and then extracting page mapping data from them. As a result, we can get a complete page mapping table for log blocks.

3.5 Reforming the Merge Operation in FAST FTL

During crash recovery, we must make two block and page mapping tables equivalent with ones just before the crash occurs. However, for the page mapping table, the recovered table may be different from the last table in main memory. This is because in-memory page mapping table can be updated due to full merge operations for victim log blocks even after a series of checkpoints are performed. The problem resulting from this-like difference is performing merge operations for invalid pages in the victim log blocks. In [4], the merge operation generates a new data block, followed by setting all pages joining in this generation with invalid on the page mapping table for log blocks. However, these invalid

LBN	PBN	MT
0	30	10
1	52	***20***
2	32	15

iv: invalid

PBN	VT	*LSN List*
520	**20**	*iv, iv, iv, iv*
501	11	*iv, 201, 202, 253*
502	12	*iv, 97, 108, 301*

(a) Merge operation before the crash.

LBN	PBN	MT
0	30	10
1	52	**20**
2	32	15

iv: invalid

PBN	VT	*LSN List*
520	**20**	*iv, iv, iv, iv*
501	11	***5***, *201, 202, 253*
502	12	***6***, *97, 108, 301*

(b) Merge operation after the crash.

Fig. 2. Reforming the Merge Operation in FAST FTL

marks cannot be reflected on already checkpointed entries of the page mapping table. So, there cannot exist invalid marks in the newly recovered page mapping table after crash recovery, resulting in useless merges on invalid pages. In order to prevent useless merges like this, we try to reform the merge operation in FAST FTL by devising two terms, VT(Victim Timestamp) and MT(Merge Timestamp). Each log blocks VT in the page mapping table points to its turn in which it will be a victim log block, while each data blocks MT in the block mapping table indicates which merge generated it. We define that MT is VT plus the number of log blocks. For example, in Fig. 1, VT=10 of the first entry in the page mapping table indicates that current log block will be tenth victim later. MT=13 of the second entry in the block mapping table means that the data block of PBN=30 was generated due to the third victim, since the number of log blocks is ten and so VT is three.

Now let us reform merge operations occurring after crash recovery by using VT and MT. Fig. 2(a) shows that before a crash occurs, the log block of PBN=500 in Fig. 1 was chosen as a victim, next a full merge operation was done for this victim log block, and finally a new log block was allocated from the free block list. The full merge operation consists of multiple individual merges, so in Fig. 1, the pages of LSN=4, 5, and 6 in all log blocks will be used in generating a new data block of LBN=1(assuming that a block has four page). Accordingly, the data block of LBN=1 in Fig. 1 allocate a new physical block of PBN=52 and also MT changes from 13 to 20, because VT of the victim log block is 10. In the page mapping table, invalid marks are made for the participants for the new data block. Let us assume that after Fig. 2(a), a system failure happened. According to our recovery strategy, two tables of Fig. 2(b) will be made. However, the page mapping table of Fig. 2(b) differs in that of Fig. 2(a). This is because invalid marks did not be reflected on the former. In case that the log block of PBN=501 in Fig. 2(b) will be selected as a victim later, useless data block will be created for LSN=5 and 6 in all log blocks. This problem can be solved by using VT and MT. Before create a new data block by performing individual merge operation like LSN=5 and 6 of Fig. 2(b), we first find the data block with the same LBN

in the block mapping table(LBN=1 in Fig. 2(a)) and then compare MT with VT. If MT is larger than VT, then we skip current individual merge operation, since this indicates that some victim already made current data block for same LSNs.

4 Evaluation

Runtime Overhead for Mapping Management. In this section, we evaluate the mapping management overhead using the total write overhead between LTFTL based FAST scheme and our FAST scheme under the OLTP workload.

In the case of LTFTL scheme in the FAST FTL, each N_{lt-log} logs will be written as an mapping log entry. Moreover, if the total log size exceeds the $N_{lt-threshold}$, FTL must store the whole mapping table consisted with N_{table} pages. Therefore, $Cost_{LTFTL}$, the write overhead of LTFTL based FAST, is as follow:

$$Cost_{LTFTL} = \frac{N_{log}}{N_{lt-log}} + \left\lfloor \frac{N_{log}}{N_{lt-threshold}} \right\rfloor \times N_{table} \tag{1}$$

If the FAST FTL uses FASTcheck scheme, each N_{latest} merge operation will write a page as the N_{latest} block's mapping table in the FASTcheck block. In this case, $Cost_{FASTcheck}$, the write overhead cost using FASTcheck, is hear:

$$Cost_{FASTcheck} = \frac{N_{merge}}{N_{latest}} \tag{2}$$

From this notation, we calculated the mapping management cost between the LTFTL based FAST and FASTcheck based FAST. Under the total 3,594,850 write operation, $Cost_{LTFTL}$ was 30,544 and $Cost_{FASTcheck}$ was 3,511.

Recovery Overhead. At the recovery phase, any FTL scheme must read the address table and its changed table log. In our approach, FTL has two phase read operation as we mentioned at Section 3.4. The N_{table} FASTcheck area pages must be read at the first read phase, and average $(N_{latest} \times N_{page})/2$ pages has to be read for the uncheckpointed information. The LTFTL approach also read N_{table} table pages, $(N_{lt-threshold}/N_{lt-log})/2$ log pages, and $N_{lt-lot}/2$ pages' spare area at the recovery time. However the $N_{lt-threashold}$ would be increased if the log area size became large, our approach has suitable read count for the recovery operation.

5 Conclusion

In this paper, we proposed an efficient power-off recovery scheme for a hybrid FTL scheme, FAST. Instead of writing the new address mapping information for every page write operation, our FASTcheck scheme checkpoints the address mapping information periodically, and can recover the mapping information which is

generated since the last checkpoint. As shown in Section 4, the saving in writing the metadata during the normal mode in FASTcheck far outweighs.

We plan three future works. The first one is to implement our scheme in real boards and to verify its efficient. The second is to optimize the excessive read overhead in incremental recovery phase. And finally, we will investigate whether our scheme can be applicable to other FTLs.

References

1. Ban, A.: Flash file system, US Patent 5, 404, 485 (April 4, 1995)
2. Ban, A.: Flash file system optimized for page-mode flash technologies, US Patent 5, 937, 425 (August 10, 1999)
3. Chung, T.-S., Lee, M., Ryu, Y., Lee, K.: PORCE: An efficient power off recovery scheme for flash memory. J. Syst. Archit. 54(10), 935–943 (2008)
4. Kim, J., Kim, J.M., Noh, S.H., Min, S.L., Cho, Y.: A space-efficient flash translation layer for CompactFlash systems. IEEE Transactions on Consumer Electronics 48(2), 366–375 (2002)
5. Lee, S.-W., Park, D.-J., Chung, T.-S., Lee, D.-H., Park, S., Song, H.-J.: A log buffer-based flash translation layer using fully-associative sector translation. ACM Trans. Embed. Comput. Syst. 6(3), 18 (2007)
6. Sun, K., Baek, S., Choi, J., Lee, D., Noh, S.H., Min, S.L.: LTFTL: lightweight time-shift flash translation layer for flash memory based embedded storage. In: Proceedings of the 8th ACM International Conference on Embedded Software, pp. 51–58. ACM, New York (2008)

Time-Predictable Computing*

Raimund Kirner[1] and Peter Puschner[2]

[1] Department of Computer Science
University of Hertfordshire, United Kingdom
r.kirner@herts.ac.uk
[2] Institute of Computer Engineering
Vienna University of Technology, Austria
peter@vmars.tuwien.ac.at

Abstract. Real-time systems need to be *time-predictable* in order to prove the timeliness of all their time-critical responses. While this is a well-known fact, recent efforts of the community on concretizing the predictability of task timing have shown that there is no common agreement about what the term *time-predictability* exactly means. In this paper we propose a universal definition of time-predictability that combines the essence of different discussions about this term. This definition is then instantiated to concrete types of time-predictability, like worst-case execution time (WCET) predictability. Finally, we introduce the concept of a timing barrier as a mechanism for constructing time-predictable systems.

1 Introduction

Research on real-time computing is an established field with several decades of research. Time-predictability is exactly one of the preconditions necessary to verify the correct operation of a real-time system. However, except for some recent efforts, there was no visible attempt to precisely define what time-predictability, characterizing the temporal behavior of tasks, exactly means. This might be explained by the fact that traditional research on real-time computing was based on design choices where the available components were (implicitly) sufficiently time-predictable. However, with the increasing performance gap between processing speed and memory access times, the commercial computer components available on the market have become more complicated, as they introduced peak-performance enhancing features like pipelines and caches. With the currently observed stall on maximal performance of single-core processors, the situation again has become more challenging for timing analysis – nowadays we observe a strong move towards the use of multi-core systems in which the competition for memories that are shared among the processor cores negatively affects predictability.

* The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7,2008-2011] under grant agreement no 214373 (ArtistDesign, http://www.artist-embedded.org/) and the IST FP-7 research project Asynchronous and Dynamic Virtualization through performance ANalysis to support Concurrency Engineering (ADVANCE).

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 23–34, 2010.

Analysis of the worst-case execution time (WCET) [1] for systems built from conventional components is relatively complicated [2,3]. On one hand it is very difficult to re-engineer the behavior of a processor in order to construct a model that reflects the timing of all operations of the processor correctly. On the other hand high computational effort is needed to calculate the WCET bound.

To react on the current hardware trend towards computer systems with increasingly complex temporal behavior, the timing analysis community started to work on simpler hardware components for which WCET analysis can be done with reasonable effort. To systematize this effort, it is advantageous to first work out a common understanding of *time-predictability*. This seems to be important as we observed that the term time-predictability has been used with different meanings in the timing-analysis community so far. To give an example, Thiele et al. [4] define time-predictability as the pessimism of WCET analysis and BCET analysis, while Grund defines time-predictability as the relation between BCET and WCET [5].

In this paper we contribute a new view on time-predictability of real-time tasks. The definition of the term time-predictability is based on a timing model used for WCET analysis. Besides time-predictability, we also discuss the efforts needed to construct such a model, as well as the necessary analysis efforts and the stability of the obtained timing behavior.

2 Worst-Case Execution Time Analysis

One of the main reasons for writing this paper is that we observe that there are different, even divergent, understandings of what *predictable task timing* is meant to stand for (see above).

Task timing analysis and WCET analysis are just a part of the software development process for dependable hard real-time systems. The whole software development process is hierarchical, in order to make the design and implementation of complex applications simple.

The higher levels of the development process deal with the decomposition of the application software into groups of tasks and finally single tasks, they assign tasks to the hardware components of the computer system, and they construct and check schedules that fulfill the communication and synchronization requirements of the tasks in the application. The task level of software development has to make sure that the tasks deliver the correct functionality and meet their execution-time constraints. To achieve the latter, the timing of tasks, in particular their WCET, should be easy to assess, i.e., task timing should be in some way forseeable.

The task interface separates the task level from the higher levels of software development. It characterizes the types, possible values and the meaning of the inputs and outputs, and the semantics and execution time of the task. To support the hierarchical development, a task system needs to be (nearly) composable, i.e., the task interfaces have to be small and the interactions of the tasks via the interface should be weak [6] – strong interactions or dependencies via an interface would either necessitate a complex task-wise analysis that took these these dependencies into account or, alternatively,

would yield highly pessimistic results when combining the results of the analyses of the individual tasks.

In accordance to the above-mentioned ideas the tasks subject to WCET analysis are generally simple tasks (see [7] for the description of the simple-task model). Simple tasks have a plain *input – processing – output* structure, where all inputs are available (and read) at the beginning, outputs are available (and written) at the end, and the central part of the task is free of any I/O, task synchronization, waiting, or delays.

In the following we understand a task as an object that is defined by its code and the hardware on which the code is expected to run. We assume that a task executes without any disturbances by other tasks or system activities. Task interactions happen via the tasks data interface (consisting of inputs from the environment and the values of its state variables at the beginning of its execution), and via the state of the hardware at the task start time which might differ between different executions. Note that we assume that the state of a task – the set of state variables that keep their values between successive executions – is part of its input. Stateful tasks are modeled by tasks that send the values of their state variables at the end of each execution and read them in again at the beginning of their next execution. Making the state of a task part of its input has the advantage that task timing can be modeled locally, without the need to model how the task state might develop over a number of task instantiations.

Possible Understandings of Temporal Predictability

The term *temporal predictability* respectively *time predictability* can be understood in different ways, depending on which questions about the temporal behavior of tasks one wants to answer. The different possible understandings of the term can be grouped into three categories.

- Predictability as a statement describing the properties of the phenomenon execution time of a task, e.g., the possible range or possible occurrence patterns of execution times of the task.
- Predictability as a statement about the properties of the process of modeling the timing behavior of a task. Notions of this type might be used to answer questions about how difficult it is to build or evaluate a timing model for some code running on some target processor, or what the cost of constructing such a model are.
- A combination of the above.

Predictability as Statement about the Phenomenon Execution Time

In this section we present some interpretations of time predictability when this term is meant to characterize properties of the phenomenon execution time of a task, assuming an undisturbed execution of the task and assuming a given task software and target hardware.

- "For given input data and a given hardware state at the time the task starts the execution time of an execution is determined." – predictability as a statement about the deterministic behavior of hardware and software.

- "For a given set of possible inputs and start states there is a minimum execution time (BCET) and a maximum execution time (WCET) that define an interval [BCET, WCET] in which all possible execution times of the task can be found." – predictability as a measure for the variability of execution times. Obviously, a smaller interval yields a smaller expected error for "guesses" about the execution times of executions. Thus the smaller the interval [BCET, WCET] the better the predictability in this understanding. The special case that BCET = WCET yields the highest predictability ranking in this understanding.
- Another interpretation of predictability could aim at describing how the execution time of a task develops over time, i.e., from one execution to the next, etc. For example, think of a periodic tasks whose execution times show the repeated pattern $t1$, $t2$, $t3$. One might consider this as a very predictable execution-time pattern. While this pattern might indeed be considered predictable at a higher level, such execution-time patterns are beyond what we can argue about at the task level. Arguing about execution time patterns requires that we need to know how the inputs of the task change from one task instance to the next. This is however beyond the scope of what can be expected and done at the task level. When considering a task – purely at the task level, i.e, independent of how inputs evolve – arguing about the timing dynamics over time does not make sense.

The following sections of the paper deal with the understandings of timing predictability that also incorporate the properties of the execution-time modeling process, i.e., how close the modeled timing resembles the real execution times, respectively how costly it is to build an execution-time model.

3 Time-Predictability

In the following we present a definition for predictability of the task timing. We build this definition around an abstract meaning of timing, and concretize it then for two special cases, the predictability of the worst-case execution time (WCET) and the best-case execution time (BCET). We start with a discussion of predictability in general.

3.1 Constructability of Timing Model

The construction of an adequate system model for the type of prediction one is interested in can be quite challenging, for which we identify three reasons, as shown in Figure 1:

1. It can be challenging to observe the system behavior in sufficient detail, as certain internal mechanisms might not be accessible by the provided interfaces.
2. Once we get all the relevant details that are necessary to model the timing of a system, the model might become so complex that it is hard to understand the behavior at an adequate abstraction level for the modeling purpose.
3. It has to be ensured that the model reflects the modeling purpose in a safe way, for example, by ensuring that the model provides an overapproximation of the system's possible behavior.

The level of complexity of these three steps are an indicator for the constructability of the model. While the constructability of the timing model of a system is somehow related to the predictability of the timing model, we consider these two aspects separately. Keeping constructability of predictability of the model separate is motivated by the fact that the construction of the hardware part of the model has to be done only once for each processor, while this model can subsequently be used for the prediction of each program running on that hardware.

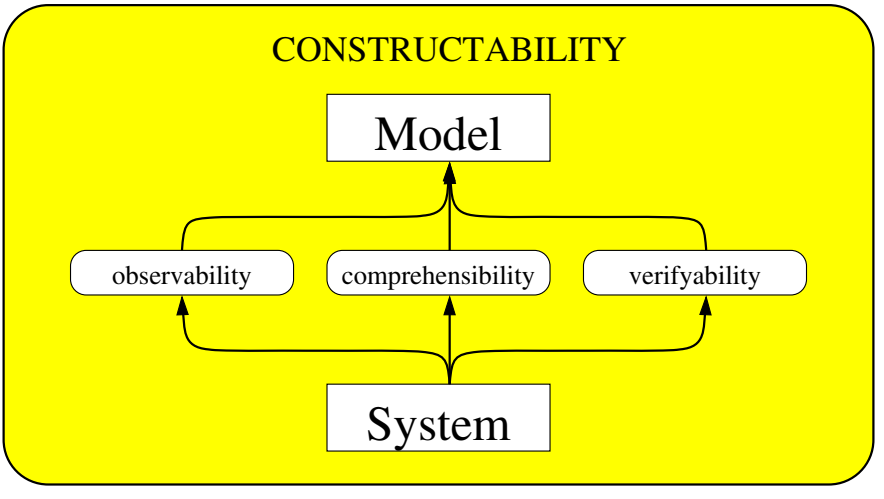


Fig. 1. Constituents for Constructability

3.2 Constituents of Predictability

When we try to predict a system's behavior we do this by analyzing a (simplified) model of the system, i.e., the model is the central object for prediction. Discussing the predictability of a system thus results in a discussion of the system model.

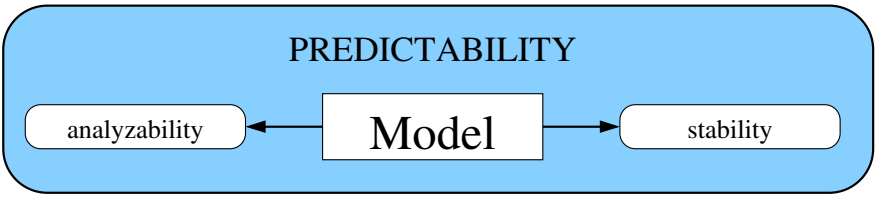


Fig. 2. Constituents for Predictability

As shown in Figure 2, the aspects of the model to be discussed for judging the system's predictability are the *analyzability* and the *stability* of the model.

Analyzability of the model: To make a prediction we have to analyze the model for the properties of interest. The analysis of the model is different from the verification of the model. Even if the correctness of the model can be ensured based on simple construction rules for which the overall correctness can be shown by induction, the analysis of the model to calculate a parameter of interest can still result in a global search problem over the whole model. The degree of analyzability depends on how much effort has to be spent to calculate the investigated property and on how accurate the obtained result is.

Stability of the model: The predictability of the model in general also depends on the degree of variation in the investigated behavior. The fewer open parameters the model has, the more exact a prediction can be made. Furthermore, the smaller the variation in the possible behavior is, the more exact a prediction can be made.

These two aspects in combination determine the level of time predictability.

3.3 Predictability of Task Timing

To describe the task timing, the system model has to contain information about the executed program instructions and the durations of these actions on the concrete hardware platform. Constructing a precise and correct timing model is typically quite challenging. This is mostly because the temporal processor behavior is not very well documented, for example, to hide intellectual properties of the hardware implementation, or simply due to the lack of the construction of a precise model during the processor design phase. It might be also the case that the processor manual simply contains errors on the processor's timing behavior. The missing level of detail in the documentation also reduces the comprehensibility of the timing results observed by experiments.

The analyzability of the timing model depends both on the complexity of the program to be analyzed and on the complexity of the hardware platform. The properties to evaluate are the accuracy obtained by the timing analysis and the computation effort needed to achieve this result. This implies that the availability of an efficient analysis method is a prerequisite for a good time-predictability.

At a first glance, defining time-predictability not only on the properties of the system but also on the properties of the available analysis methods might seem surprising. However, if we look closer and try to discuss time-predictability (or predictability in general) without considering the scope of any calculation technique, we would lose the foundation for any ordering relation between different degrees of time-predictability. What remains would be a boolean decision of whether the requested information is decidable or not (from a theoretical point of view). Therefore, the calculation cost for deriving the information of interest is of importance for a practical definition of time-predictability. An indicator for the computational effort of the analysis is the explicity of information [8]:

> *Explicity of Information:* Given a system model, the *explicity of information* refers to the computational complexity required to infer this information. While a relative explicity ordering of different information typically depends on the underlying calculation method, full explicity is given if the information of interest is directly coded in the model.

The calculation of the *stability* of the timing model will depend on the specific type of timing behavior one is interested in. For example, if one is only interested in a specific timing invariant like the worst-case execution time, the variability of the task's execution time does not play a significant role. However, if one is interested in the ability to predict temporal trends for dynamic system optimization, like, for example, the current performance bottlenecks in a system, the *variability* among the possible execution times plays a more serious role.

By instantiating the generic concept of predictability given in Figure 2 to the time domain, we get the following definition of time-predictability:

> *Time Predictability:* The *time predictability* of a system model refers to the ability of calculating the duration of actions on the concrete system. This ability is defined in the sense of tractability rather than decidability. Thus, to have a high time-predictability, the precise calculation of the system's timing behavior must not only be decidable, but also efficient calculation techniques must be available. As a consequence, the time predictability of a model can improve as soon as more efficient calculation techniques become available.

Predictability of WCET and BCET. The above definition of time-predictability gives an intuition of what is important to make a system time-predictable. However, when it comes to the point of evaluating the predictability of specific timing properties of the system, a more exact instantiation of time-predictability is needed.

For example, in the domain of safety-critical systems, typically the WCET of a task is of primary interest to prove the timeliness of the system. As discussed above, for deciding the WCET-predictability of a system, the stability of the timing model is of minor interest. As shown in Figure 3, only the pessimism of the calculated upper bound (UB) is of interest. Thus the analyzability of the system depends on the tightness of the WCET estimate. The stability of the timing model, which could be defined as the execution-time variability as shown in Figure 3, is not directly relevant for the WCET-predictability.

What has been said about the WCET-predictability holds in an analogous form for the BCET-predictability.

3.4 Related Work on Concretizing Time-Predictability

The aim of working out a suitable understanding of time-predictability is a quite recent activity. Though time-predictability has always been the idea behind the design of real-time systems, the term time-predictability was typically used without explicit introduction. Often, architectural patterns have been introduced as time-predictable mechanisms. For example, the so-called *temporal firewall* introduced by Kopetz is a push/push interface that decouples the timing behavior of the sender and the receiver of a message [9].

Within the work of Lee et al. on the PRET processor, with the fundamental idea of giving the instruction set of a processor a time semantics, they use time-predictability in close combination with repeatability of timing [10,11].

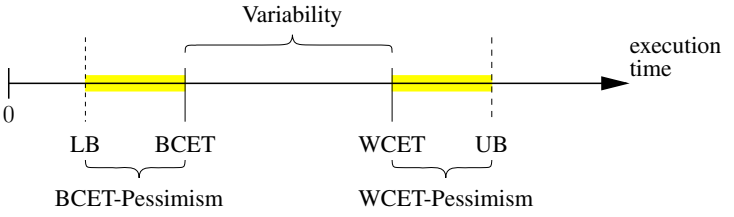


Fig. 3. The Consequences of Limited Time-Predictability

Sometimes, time-predictability was indirectly described by listing the open problems of WCET analysis and their origins. Kirner et al. have described the limitations of decomposition of WCET analysis [2]. Wilhelm et al. have identified the input-dependency and concurrence of control flow as well as the propagation of local hardware states as the major reasons for unpredictability of timing [12].

Thiele and Wilhelm have presented one of the first approaches of defining the meaning of time-predictability [4]. Translated in our terminology given in Figure 3, they described time-predictability as the pessimism of the WCET or BCET analysis. What they called "best case predictability" is the BCET-Pessimism in Figure 3, and what they called "worst case predictability" is the WCET-Pessimism in Figure 3.

A different definition of time-predictability was given by Grund [5], which was actually also the first attempt to formalize the meaning of time-predictability. The basic approach of Grund was to define the time-predictability of a program p by the relation $\frac{BCET_p}{WCET_p}$. He expressed this relation based on the set of possible system states Q and on the set of possible input values I. By denoting the execution time of a program p as $T_p(q,i)$ with $q \in Q$, $i \in I$ the time-predictability $Pr_p(I,Q)$ was defined as

$$\mathrm{Pr}_p(I,Q) \;=\; \min_{q_1,q_2 \in Q} \; \min_{i_1,i_2 \in I} \; \frac{T_p(q_1,i_1)}{T_p(q_2,i_2)} \;=\; \frac{BCET_p}{WCET_p}$$

If we translate this definition of time-predictability into our terminology given in Figure 3, it is the relative value of the execution-time variability of the program. Grund further decomposed the relative execution-time variability to the effects induced by the input data:

$$\text{State-Pr}_p(I,Q) \;=\; \min_{q_1,q_2 \in Q} \; \min_{i \in I} \; \frac{T_p(q_1,i)}{T_p(q_2,i)}$$

and to the effects induced by the hardware state:

$$\text{Input-Pr}_p(I,Q) \;=\; \min_{q \in Q} \; \min_{i_1,i_2 \in I} \; \frac{T_p(q,i_1)}{T_p(q,i_2)}$$

Compared with our definition of time-predictability, the definitions given by Thiele and Wilhelm, as well as Grund, there is a common pattern. They both represent important aspects of time-predictability, and in fact, they can be identified as some of the aspects of our definition of time-predictability. We consider the mentioned pessimism of the analysis by the *analyzability*, and the mentioned execution-time variability by the *stability* of the timing model.

3.5 On the Formalization of Time-Predictability

While the above definitions of time-predictability given by Thiele et. al and by Grund each focus on different aspects, we present a holistic definition of time-predictability that incorporates them both. With tm being the timing model that includes the hardware platform and the program to be analyzed, we get a formula for the time-predictability as follows:

$$\begin{aligned} \Pr(tm) \;=\; & analyzability(tm)^{k_a} \cdot \\ & stability(tm)^{k_s} \end{aligned}$$

To get a value for the time-predictability within the interval 0 (completely unpredictable) to 1 (maximal predictable), we would like to define the individual functions to be within the interval $0 \ldots 1$, i.e.,

$$0 \leq analyzability(tm), stability(tm) \leq 1$$

The weighting exponents k_a and k_s have to be within the range $0 \ldots \infty$. A weighting exponent of value of 1 results in a neutral weighting of the predictability ingredient, while a value smaller than 1 gives it less emphasis, and a value greater than 1 gives it more emphasis.

The stability could be defined as:

$$stability(tm) \;=\; \frac{BCET_{tm}}{WCET_{tm}}$$

The analyzability expresses the needed analysis effort as well as the achieved accuracy of the result. However, to judge the accuracy we need to define the reference result. Thus, the analyzability might only be defined for a specific instance of time-predictability. Let us in the following look at the specific instances of predictability for the WCET and BCET.

Formalization of WCET-Predictability and BCET-Predictability. The predictability of the WCET does not need to consider the stability of the timing, as it focuses only on the single WCET value. With the analog argument we can say that the predictability of the BCET does not need to consider the stability of the timing. Thus, we set the weighting exponent for the stability to zero, which indicates non-significance for the stability: $k_s = 0$. Assuming that we use the same timing model for calculating both the WCET and the BCET, we get the following formulas for WCET-predictability and BCET-predictability:

$$WCET: \; \Pr_{WCET}(tm) \;=\; analyzability_{WCET}(tm)^{k_a}$$

$$BCET: \; \Pr_{BCET}(tm) \;=\; analyzability_{BCET}(tm)^{k_a}$$

The analyzability depends on the analysis effort as well as on the accuracy of the result. As it is not obvious how to define the needed analysis effort in a way that is independent of the concrete analysis method, we ignore the analysis effort for the first and focus

only on the accuracy of the result. For the WCET-predictability it is natural to define the accuracy based on the obtained upper bound UB_{tm} of the execution time, while for the BCET-predictability we would define it based on the obtained lower bound LB_{tm}:

$$WCET: \ analyzability_{WCET}(tm) \ = \ \frac{WCET_{tm}}{UB_{tm}}$$

$$BCET: \ analyzability_{BCET}(tm) \ = \ \frac{LB_{tm}}{BCET_{tm}}$$

Summary on the Formalization of Time-Predictability. To formalize time-predictability we had to ignore the analysis effort needed to calculate the time prediction. This omission was necessary since the analysis effort can be hardly normalized, especially for different platforms and analysis methods. Furthermore, the formulas include weighting exponents to adjust the priority between analyzability and stability. For a comparison based on time-predictability it would be necessary to agree on some specific weighting exponents. Furthermore, a precise calculation of time-predictability may fail in practice as it requires to calculate precise values for the WCET and the BCET.

4 Time-Predictable Computer Systems

Given the definition of time-predictability and its instances like WCET-predictability, it is the obvious question of how systems should be built to make them time-predictable. In this section we introduce the concept of a *timing barrier*, a mechanism whose frequent use in the system design is an indicator for a good time-predictability.

First, we need to recite the concept of the *timing relevant dynamic computer state* (TRDCS), which is that part of the system's state space that has a influence on the timing behavior and that will not remain constant during all program executions [13]. Based on this, a *timing barrier* is defined as follows:

> *Timing Barrier:* The *timing barrier* is a partitioning in the time-domain of control-flow by a mechanism into two parts, called predecessor and successor, such that this mechanism reduces the fraction of the system state belonging to the predecessor that has an influence of the TRDCS belonging to the successor. Thus, a timing barrier decouples the execution time of an instruction sequence from its execution history.

This concept of a *timing barrier* was actually motivated by the concept of the *temporal firewall* as introduced by Kopetz and Nossal [9]. However, there is one fundamental difference between these concepts: while the temporal firewall decouples the execution time as well as the execution start between two programs, the timing barrier focuses only on decoupling the execution time, but not necessarily the start of an execution. In fact, the timing barrier is more a light-weight concept that is meant to be applied at many different instances within the system.

Timing barriers might be realized in software or hardware. It is important to note that a timing barrier in its generic definition does not make the execution time of the successor completely independent of the predecessor, but reduces the actual coupling. This is actually the fundamental principle of how to achieve nearly composable systems as described in Section 2. Timing barriers also tend to be useful to guide experts when selecting the hardware platform in the early design stage of a system. Even if no timing analysis exists so far, the presence of timing barriers provides hints about the level of time-predictability of the processor.

An example for a timing barrier is the flush instruction of a cache, which brings the cache into a well-defined state that is independent of the execution history. However, the timing of the successor may still depend, for example, on the pipeline state leaving the predecessor.

There are many more examples of timing barriers, and it would make great sense to collect and categorize them in order to get a better understanding of how to build time-predictable systems. Thus, we will collect concrete instances of timing barriers in our future work. By doing so we hope to get a better understanding about how to construct systems with a good time-predictability. It would also be interesting to investigate to what extent the availability of timing barriers can help to judge the time-predictability of hardware platforms.

5 Summary and Conclusion

Real-time systems have to be time-predictable to ensure their timeliness. Despite this importance of time-predictability, there is still no common agreement of what time-predictability of tasks exactly means. In this paper we therefore reviewed different definitions of time-predictability proposed so far. For our view of time-predictability we have focused on how good a timing model can get. We distinguish between the constructability of such a timing model and the time-predictability of the timing model with respect to the real system behavior. We proposed a new definition of time-predictability of code that combines the essence of previous definitions, namely the analyzability and the stability of the timing model. Further, we have also presented a formalization of this definition.

As a first proposal of how to inspect real hardware for time-predictability, we introduced the concept of a *timing barrier*. Timing barriers describe mechanisms of architectures that limit the propagation of timing effects. Using hardware patterns that implement timing barriers will help to reduce the complexity of timing analysis, which is a prerequisite for constructing systems whose timing is easily understandable and predictable.

Acknowledgments

We would like to thank the numerous colleagues who participated in the discussions about time-predictability and provided valuable feedback to our work, like, for example, at the RePP'2009 workshop in Grenoble.

References

1. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckman, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenstrom, P.: The worst-case execution time problem - overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS) 7(3) (April 2008)
2. Kirner, R., Puschner, P.: Obstacles in worst-cases execution time analysis. In: Proc. 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Orlando, Florida, pp. 333–339 (May 2008)
3. Schoeberl, M.: Time-predictable computer architecture. EURASIP Journal on Embedded Systems (2009)
4. Thiele, L., Wilhelm, R.: Design for Timing Predictability. Real-Time Systems 28, 157–177 (2004)
5. Grund, D.: Towards a formal definition of timing predictability. In: Workshop on Reconciling Performance with Predictability, Grenoble, France (October 2009)
6. Simon, H.A.: The Architecture of Complexity. In: The Science of the Artificial, 1st edn., pp. 192–229. MIT Press, Cambridge (1969) ISBN: 0-262-69023-3
7. Kopetz, H.: Real-Time Systems - Design Principles for Distributed Embedded Applications. Kluwer, Dordrecht (1997) ISBN: 0-7923-9894-7
8. Wögerer, W.: The loss of explicity for static WCET analysis during compilation. Master's thesis, Technische Universität Wien, Vienna, Austria (2003)
9. Kopetz, H., Nossal, R.: Temporal firewalls in large distributed real-time systems. In: 6th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems (1997)
10. Edwards, S.A., Lee, E.A.: The case for the precision timed (pret) machine. In: Proc. 44th Design Automation Conference (DAC), San Diego, California (June 2007)
11. Edwards, S.A., Kim, S., Lee, E.A., Liu, I., Patel, H.D., Schoeberl, M.: A disruptive computer design idea: Architectures with repeatable timing. In: Proc. IEEE International Conference on Computer Design. IEEE, Los Alamitos (2009)
12. Wilhelm, R., Ferdinand, C., Cullmann, C., Grund, D., Reineke, J., Triquet, B.: Designing predictable multi-core architectures for avionics and automotive systems. In: Workshop on Reconciling Performance with Predictability, Grenoble, France (October 2009)
13. Kirner, R., Kadlec, A., Puschner, P.: Precise worst-case execution time analysis for processors with timing anomalies. In: Proc. 21st Euromicro Conference on Real-Time Systems, Dublin, Ireland, pp. 119–128. IEEE, Los Alamitos (July 2009)

OTAWA: An Open Toolbox for Adaptive WCET Analysis

Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat

Institut de Recherche en Informatique de Toulouse
University of Toulouse, France
{ballabri,casse,rochange,sainrat}@irit.fr

Abstract. The analysis of worst-case execution times has become mandatory in the design of hard real-time systems: it is absolutely necessary to know an upper bound of the execution time of each task to determine a task schedule that insures that deadlines will all be met. The OTAWA toolbox presented in this paper has been designed to host algorithms resulting from research in the domain of WCET analysis so that they can be combined to compute tight WCET estimates. It features an abstraction layer that decouples the analyses from the target hardware and from the instruction set architecture, as well as a set of functionalities that facilitate the implementation of new approaches.

Keywords: Real-time, Worst-Case Execution Time, static analysis.

1 Introduction

In hard real-time systems, critical tasks must meet strict deadlines. Missing one such deadline may have dramatic consequences, either in terms of human lives or economical and environmental issues. To avoid this, special attention is paid on determining a safe schedule for tasks.

Real-time task scheduling has been and is still a hot research topic, but it relies on the knowledge of the Worst-Case Execution Time (WCET) of the critical tasks. While this time is generally considered as known, the techniques required to compute it are not straightforward and cannot always support complex target architectures. As a result, research on WCET analysis is also very active.

The general approach to WCET analysis includes three steps: (a) flow analysis mainly consists in identifying (in)feasible paths and bounding loops; (b) low-level analysis aims at determining the global effects of the target architecture on execution times and at deriving the worst-case execution times of code snippets; (c) finally, the results of the flow and low-level analyses are combined to derive the overall WCET. For each of these three steps, several techniques have been investigated in the literature. Besides, several tools have been developed to implement these results. Most of them are still research prototypes but a few are now commercialized.

The design of the OTAWA toolbox presented in this paper started in 2004, at a time where very few tools were publicly available. Our main motivation was

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 35–46, 2010.

to provide an open framework that could be used by researchers to implement their analyses and to combine them to already implemented ones. The expected characteristics of this framework were versatility (ability to support various target hardware configurations as well as various instruction sets) and modularity (support to facilitate the implementation of new analyses). OTAWA is not a tool but a *toolbox*: this means that it comes as a C++ library that can be used to develop WCET analysis tools. However, it includes a number of algorithms which make this task really easy (example tools are distributed with the library).

The paper is organized as follows. Section 2 reviews the state of the art on WCET analysis and gives a short overview of existing tools. The OTAWA toolbox is presented in Section 3 and Section 4 illustrates its capacities through a short report of how it has been used in several projects. Concluding remarks are given in Section 5.

2 Worst-Case Execution Time Analysis

2.1 Different Approaches to WCET Analysis

The execution time of a program on a given processor depends on the input data and on the initial state of the hardware (mainly on the contents of the memories). To evaluate its worst-case execution time (WCET), it is necessary to consider all the possible input values and all the possible initial hardware states. In most of the cases, this is not feasible: (a) determining input data sets that cover all the possible execution paths is hard, especially with floating point inputs[1]; (b) even if this was possible, the number of paths to explore would be too large and the time required to measure all of them would be prohibitive; (c) it is not always possible to initialize the hardware state prior to measurements so as to investigate all the possible states. Because it is not feasible to measure all the possible paths considering all the possible initial hardware states, it is now commonly admitted that WCET analysis must break down the execution paths into code snippets (generally basic blocks) so that time measurement or estimation is done on these units and the overall WCET is computed from the unit times.

One way to classify approaches to WCET analysis is to consider the way they determine the worst-case execution time of code snippets. Some of them are based on measurements performed preferably on the real target hardware, but possibly on a simulator [4] while other ones use a model of the hardware to compute these times [16] [19] [27].

Another approach is to focus on the way the unit times are combined to derive the complete WCET of the task. Some approaches need the program under analysis to be expressed in the form of an Abstract Syntax Tree (AST) [20] [10] and compute the WCET using analytical formulae related to the algorithmic-level statements found in the code. These approaches do not fit well with codes that

[1] An exception is the case of programs written under the single-path programming paradigm [24] but this still remains marginal.

have been optimized at compile time. Other solutions consider the Control Flow Graph (CFG) built from the object code and use path-based calculation [15] or formulate the search of the longest execution path as an Integer Linear Program [18].

Besides to these contributions that are at the root of research on WCET analysis, a lot of work has been done to design algorithms able to derive execution times of basic blocks considering various and more and more complex hardware features. These solutions will be shortly reviewed below.

2.2 Analysis of the Behavior of Hardware Mechanisms

WCET estimates are expected to be safe since the determination of a schedule of tasks that insures that hard deadlines will be met is based on them. So far, no way to obtain guaranteed WCET upper bounds without getting into the hardware architecture details has been found. Instead, existing methods compute execution time with cycle-accuracy. As a result, a number of papers have been published to show how to take into account the behavior of specific variants of hardware mechanisms in WCET analysis. In addition, it is often required for WCET estimates to be tight so as to avoid oversizing the hardware to insure schedulability. For this reason, research work also focuses on reducing the WCET overestimation.

Many hardware schemes now present in the processors used in embedded systems are supported by WCET analysis techniques:

- the processor pipeline has been studied for many years from the first scalar architectures [20] to superscalar configurations [21] [30] and pipelines featuring dynamic instruction scheduling [16] [19] [27];
- the analysability of cache memories has also been largely investigated. Several techniques have been proposed to analyze the behavior of instructions caches [1] [15], data caches [29] [28], and multi-level memory hierarchies [14];
- dynamic branch predictors have retained attention too [9] [3] [6];
- other mechanisms like the memory row buffer have been considered [5] to model specific hardware.

A recent paper [31] reviews and compares the WCET tools that have been designed by members of the ARTIST European Network of Excellence. Some of them are research prototypes and other ones are commercial tools. Since most of these tools were not publicly available or not mature enough at the time we started research analysis (2004), designing our own software was a necessity. However we always kept in mind the need of making it open so that it can host any research result on WCET analysis.

3 The OTAWA Toolbox

3.1 Objectives

Developments for the OTAWA toolbox started in 2004 with the objective of designing a framework that could integrate various kinds of methods related to WCET

analysis and support a large range of target architectures. Such a framework is expected to:

- capitalize results of research on WCET estimation
- make it possible to compare two different techniques that address the same problem (e.g. instruction cache analysis) though experiments carried out under identical conditions
- support the investigation of new algorithms by allowing experiments that combine them to state-of-the-art techniques. This avoids having to make either overwhelmingly optimistic or pessimistic assumptions for features that are out of the scope of the current work.

To achieve these goals, `OTAWA` exhibits two key features. First, both the target hardware and the code under study are processed through an abstract layer: this way, analyses are designed independently from the hardware and from the instruction set and can then be used for any architecture. Second, the toolbox provides all the facilities to implement new analyses with limited effort

The overall structure of the `OTAWA` toolset is shown in Figure 1. It includes a number of components that provide two classes of facilities: some of them maintain information about the application under analysis and about the target hardware architecture, as detailed in Section 3.2; other ones manipulate and enrich this information towards an estimation of the worst-case execution time (see Section 3.3).

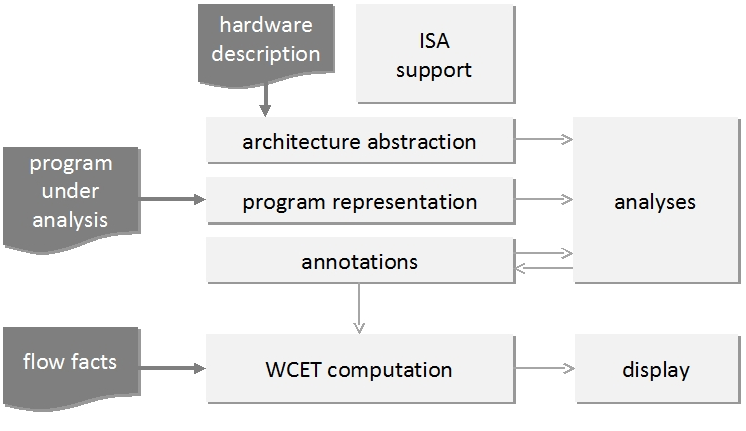


Fig. 1. Overview of the `OTAWA` framework

3.2 Abstract Layers

As explained above, `OTAWA` is organized in independent layers that provide an abstraction of the target hardware and associated Instruction Set Architecture (ISA) as well as a representation of the binary (`.elf`) code under analysis.

ISA Support. This layer hides the details of the target ISA to the rest of the toolset and provides a unique interface to retrieve the information needed for the analyses related to WCET estimation. This way, most of the components of the toolbox can be designed independently of the instruction set. The layer features support for various ISAs implemented through a set of plugins. Available plugins have been generated using our `GLISS` tool [25] that will be presented in Section 3.4 and support the following ISAs: PowerPC, ARM, TriCore, HCS12, Sparc.

Abstraction of the Hardware Architecture. Precise estimation of the worst-case execution time of instruction sequences requires a detailed knowledge of hardware parameters:

- processor: width and length of the pipeline, number of functional units and their latencies, binding of instruction categories to the functional units, specification of the instruction queues (location, capacity)
- caches: capacity, line width, organization (number of sets, number of ways), replacement policy, write policy, etc.
- memories: as part of the French MORE project, we have implemented support for various memory components (DRAM, scratchpad), each exhibiting a specific access latencies and buffering policies.

Support for additional components or new architectural features can be added by the developer with limited effort. From the user side, the hardware parameters can be specified with an XML file. Since investigating formats that could describe any kind of processor was out of the scope of our research, we decided to limit our XML format to the minimum needed to specify standard architectures. Taking into account real-life processors still requires specific developments by the user, but this is made easy by the classes and functionalities available through the `OTAWA` library.

Properties. One key facility in the `OTAWA` toolbox is the possibility of defining *properties* that can be used to annotate any kind of object defined in the library (e.g. an instruction) in a very convenient way.

These annotations are first used to build the CFG (or AST) representation of the program. The set of instructions in the program is retrieved through the ISA support layer, and a Control Flow Graph (CFG) can be built for each function. It consists of basic blocks, i.e. sequences of instructions with single entry and exit points, and of edges that express the possible flow from one basic block to another one. In the case of indirect branch instructions (e.g. implementation of a `switch` statement), it may happen that `OTAWA` requires help from the user who must specify the possible branch targets so that appropriate edges can be included in the CFG. The program under analysis is then represented within `OTAWA` as a set of interconnected CFGs.

The properties are also useful to store the results produced by the different analyses (e.g. the instruction cache analyzer annotates instructions to indicate

whether they will hit or miss in the cache and the pipeline analyzer takes this information into account to determine the instruction fetch latencies).

3.3 Analyses

As explained before, the determination of the WCET of a task involves a number of analyses that must be performed in an appropriate order. In `OTAWA`, such analyses are implemented as *Code Processors*: a code processor is a function that uses available annotations and produces new annotations.

The distributed `OTAWA` library includes a set of analyses. Some are related to flow analysis and produce information that are useful to other analyses. Examples in this category are a CFG builder, a CFG virtualizer that produces a virtual CFG with the functions inlined (this allows call-contextual analyses) or a loop analyzer that determines loop headers and dominance relationships which are used e.g. by the instruction cache analysis. Other code processors provide facilities to use state-of-the-art analysis techniques, like Abstract Interpretation [11]. Finally, a number of analyses compute data that can be combined to determine the final WCET. They include the pass that generates the integer linear program defined by the IPET method [18] but also several analyses that take into account the features of the target hardware. At this state, `OTAWA` provides the code processors needed to analyze:

- instruction caches: the approach is based on Abstract Interpretation and is close to Ferdinand's method [13]. However, our implementation includes the improvements proposed in [2] to reduce complexity while performing an accurate persistence analysis in the case of loop nests.
- pipelines: the method implemented in `OTAWA` to compute the worst-case execution times of basic blocks is original and has been presented in [27]. While the `aiT` tool uses abstract interpretation to build the set of possible processor states in input/output of each basic block, which is likely to be time consuming, our method is based on execution graphs and involves an analytical calculus. This is similar to the approach used in the `Chronos` tool [19] except for we consider a parametric view of the processor state at the entry of each basic block when they consider the worst-case processor state. Experiments have shown that our algorithm provides more accurate results with comparable computation times.
- dynamic branch predictors: we have developed an algorithm to take into account the behavior of a dynamic branch predictor [6]. It formulates the question of determining whether a branch will be always/sometimes/never well predicted as an integer linear program. However, experiments have shown that the complexity of this program is significant.

Handling Analyses Dependencies. When considering the full process of computing the WCET of a task, it appears that the involved analyses are interdependent through a producer-consumer scheme: annotations produced by one analysis are required by other passes. To respect these dependencies, the analyses

must be invoked in an appropriate order as illustrated on the example shown in Figure 2. OTAWA provides a means to automatically handle dependencies: *features* can be used to express the properties that are required or produced by a code processor. For each defined feature, a default code processor must be specified: it will be executed when the feature is required if it has still not been produced. As a result, a simple WCET tool can be designed by only invoking a final WCET computation code processor: default code processors will be automatically called to generate the missing data.

```
Object code loader
Flow facts loader
CFG builder
Loop analyzer
Instruction cache analyzer
Basic block timing analyzer
Structural constraints builder (IPET)
Cache-related constraints builder (IPET)
WCET computation (call to ILP solver)
```

Fig. 2. An example scenario for WCET estimation

Display Facilities. OTAWA provides facilities to dump out program representations and annotations produced by the different analysis. These outputs are as useful to the WCET tool developer, as well as to the real-time application developer. The former can use them to debug or to tune new analyses while the latter can get a better understanding of the program behavior and locate program regions that break the deadlines.

More recently, OTAWA has been integrated in the Eclipse programming environment. The OTAWA plugin allows benefiting from the Eclipse graphical user interface to improve OTAWA use ergonomics and helps in achieving a better integration in the development cycle. The developer can seamlessly develop an application, compile it and get the WCET of some program parts. From the graphical program representation, time-faulty program regions can be also easily identified and fixed. Figures 3 and 4 show how the source code and the CFG of the program under analysis can be displayed and colored to highlight the critical paths (the darker code regions are those that are responsible for the larger part of the total WCET).

Cycle-Level Simulator. In addition to the WCET-related analyses, the OTAWA toolbox also includes a code processor that builds and runs a cycle-level simulator. This simulator is generated on top of the SystemC library and matches the XML description of the target architecture. It makes it possible to observe the execution times related to given input values and then to get an empirical insight into the range of WCET overestimation.

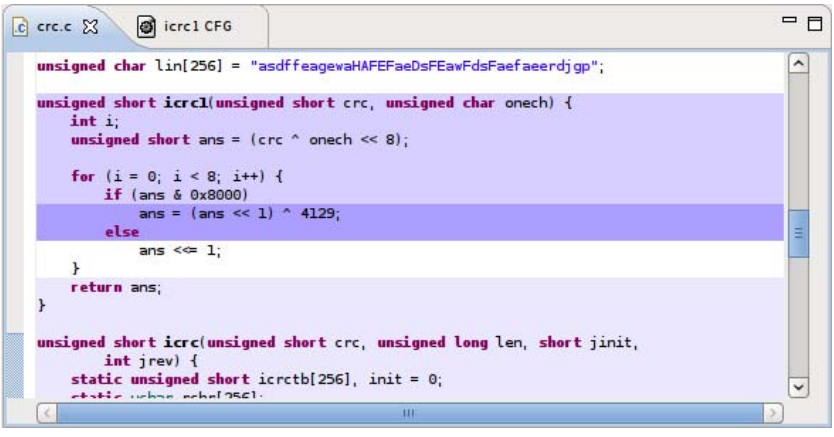


Fig. 3. Colored display of source code

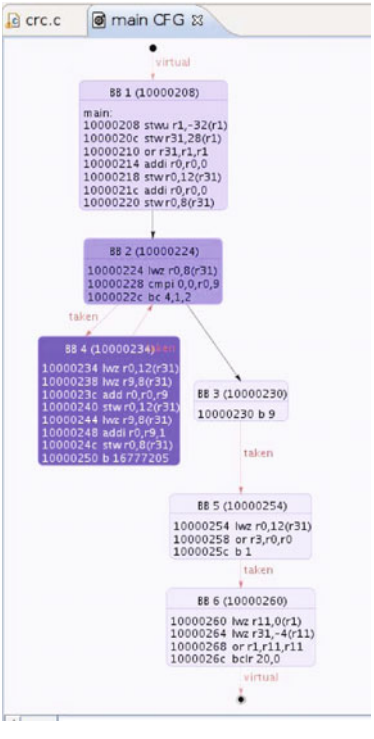


Fig. 4. Colored display of a CFG

3.4 Complementary Tools

In OTAWA, the abstraction of the Instruction Set Architecture is provided by a loader module in charge of loading the object code of the application under study and of building a representation of this code that is independent of the ISA. To design these loader modules, we use instruction set simulators generated by our GLISS tool from ISA specifications expressed in the SimNML language [25]. An instruction set simulator is a library that provides functions to decode, disassemble and emulate instructions (emulation is performed on a logic state of the memory and registers).

Another complementary tool is oRange that is used to determine flow facts and more particularly loop bounds [12]. It works on the C source code of the application and uses flow analysis and abstract interpretation techniques to derive contextual loop bounds (a loop in a function can have different bounds for different calls). Debug information inserted by the compiler in the object code are used by OTAWA to assign source-level loop bounds to the corresponding machine-level instructions. In the future, oRange will be integrated into the OTAWA framework.

Finally, the integer linear programs generated as part of the IPET method [18] are solved by invoking the lp_solve tool [32].

4 Examples of Use

The OTAWA toolbox has been involved in several projects. In this Section, we give insight into how it has been successfully used to fulfill a variety of objectives.

The goals of the MasCotTE project[2] were to investigate the possibility for WCET analysis techniques to consider off-the-shelves processors with different levels of complexity. The basic functionalities of OTAWA coupled to our GLISS tool made it possible to model two processors used in automotive applications: the Freescale 16-bit Star12X and the high-performance Freescale MPC5554. This work has been reported in [8].

In the MERASA project[3], the objective is to design a multicore processor able to execute mixed-critical workloads while offering timing predictability to hard real-time tasks. Two approaches to WCET analysis are considered: measurement-based techniques, with the RapiTime tool and static analysis, with the OTAWA toolset. Abstractions of the MERASA multicore and support for the TriCore ISA have been developed, as well as models for the specific components of the architecture: dynamic instruction scratchpad [22], data scratchpad used for stack data, predictable bus and memory controller [23]. In addition, OTAWA has been used to analyze the WCET of a parallel 3D multigrid solver and used

[2] **Ma**îtrise et **Con**trôle des **T**emps d'**E**xécution (*Controlling Execution Times*). This research has been partially funded by the French National Research Agency (ANR).

[3] **M**ulti-Core **E**xecution of Hard **R**eal-Time **A**pplications **S**upporting **A**nalysability. This research is partially funded by the European Community's Seventh Framework Programme under Grant Agreement No. 216415.

as a pilot study for the project. The contribution of this work, reported in [26], is in (a) the analysis of the synchronizations between parallel threads and, (b) the tight evaluation of the synchronization-related waiting times, based on the ability of OTAWA to analyze the WCET of specified partial execution paths.

The MORE project[4] aims at providing a framework for the investigation of code transformations used to improve several criteria like code size, energy consumption or worst-case execution time. This framework is being designed using the OTAWA toolbox and includes evaluation tools (for worst-case execution times or energy consumption), transformation tools (emulators for code compression and data placement in memories, a plugin to control GCC optimizations through the GCC-ICI interface [17]) and an iterative transformation engine that explores the transformation space to determine the best combination of transformations with respect to the requirements. The original usage of OTAWA lies in the design of transformation emulators that use the annotation system. For example, code compression is implemented the following way: profiling data is collected using the cycle-level simulator available in OTAWA; the instructions that should be compressed are determined; then each instruction is annotated with the address where it would be found in the compressed code; finally, the WCET is analyzed considering the addresses in the compressed code (the addresses are used for the instruction cache analysis). This way, it is possible to estimate the impact of compression algorithms without having to generate the real compressed code neither the corresponding code loader for WCET analysis. Further details can be found in [7].

5 Conclusion

Safe scheduling of critical tasks in hard real-time systems requires having knowledge of their Worst-Case Execution Times. WCET analysis has been a research topic for the last fifteen years, covering various domains from flow analysis (typically determining loop bounds and infeasible paths) to low-level analysis (computing the execution times of basic blocks taking into account the architecture of the target hardware). Several research groups have designed their own WCET tool to support their research and all these tools are complementary but redundant to a certain extent. It would probably be fruitful to promote their interoperability and we believe that the OTAWA toolbox presented in this paper exhibits interesting features for this purpose. First of all, it provides an abstract interface to the program to be analyzed and to the target hardware, which make it possible to develop platform-independent analyses. Second, it features a number of facilities that allow fast implementation of new analyses with a powerful means to capitalize their results towards the final WCET computation. It also provides an efficient mechanism to handle the dependencies between analyses.

[4] **M**ulti-criteria **O**ptimizations for **R**eal-time **E**mbedded systems. This research is partially funded by the French National Research Agency (ANR) under Grant Agreement ANR-06-ARFU-002.

OTAWA has been successfully used in several projects and has proved its flexibility and openness, as well as the efficiency of its key features.

The OTAWA toolbox is available under the LGPL license from www.otawa.fr.

References

1. Alt, M., Ferdinand, C., Martin, F., Wilhelm, R.: Cache Behavior Prediction by Abstract Interpretation. In: Static Analysis Symposium (1996)
2. Ballabriga, C., Cassé, H.: Improving the First-Miss Computation in Set-Associative Instruction Caches. In: Euromicro Conf. on Real-Time Systems (2008)
3. Bate, I., Reutemann, R.: Efficient Integration of Bimodal Branch Prediction and Pipeline Analysis. In: IEEE Int'l Conference on Embedded and Real-Time Computing Systems and Applications (2005)
4. Bernat, G., Colin, A., Petters, S.: pWCET a Toolset for automatic Worst-Case Execution Time Analysis of Real-Time Embedded Programs. In: 3rd Intl Workshop on WCET Analysis (2003)
5. Bourgade, R., Ballabriga, C., Cassé, H., Rochange, C., Sainrat, P.: Accurate analysis of memory latencies for WCET estimation. In: Int'l Conference on Real-Time and Network Systems (2008)
6. Burguière, C., Rochange, C.: On the Complexity of Modelling Dynamic Branch Predictors when Computing Worst-Case Execution Times. In: ERCIM/DECOS Workshop on Dependable Embedded Systems (2007)
7. Cassé, H., Heydemann, K., Ozaktas, H., Ponroy, J., Rochange, C., Zendra, O.: A Framework to Experiment Optimizations for Real-Time and Embedded Software. In: Int'l Conf. on Embedded Real Time Software and Systems (2010)
8. Cassé, H., Sainrat, P., Ballabriga, C., De Michiel, M.: Experimentation of WCET Computation on Both Ends of Automotive Processor Range. In: Workshop on Critical Automotive Applications: Robustness and Safety (2010)
9. Colin, A., Puaut, I.: Worst Case Execution Time Analysis for a Processor with Branch Prediction. Real-Time Systems Journal 18(2) (2000)
10. Colin, A., Puaut, I.: A Modular and Retargetable Framework for Tree-based WCET Analysis. In: Euromicro Conference on Real-Time Systems (2001)
11. Cousot, P., Cousot, R.: Abstract Interpretation - A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: ACM Symp. on Principles of Programming Languages (1977)
12. De Michiel, M., Bonenfant, A., Cassé, H., Sainrat, P.: Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In: IEEE Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (2008)
13. Ferdinand, C., Martin, F., Wilhelm, R.: Applying compiler techniques to cache behavior prediction. In: ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems (1997)
14. Hardy, D., Puaut, I.: WCET analysis of multi-level non-inclusive set-associative instruction caches. In: IEEE Real-Time Systems Symposium (2008)
15. Healy, C., Arnold, R., Mueller, F., Whalley, D., Harmon, M.: Bounding pipeline and instruction cache performance. IEEE Transactions on Computers 48(1) (1999)
16. Heckmann, R., Langenbach, M., Thesing, S., Wilhelm, R.: The Influence of Processor Architecture on the Design and the Results of WCET Tools. Proceedings of the IEEE 91(7) (2003)

17. Huang, Y., Peng, L., Wu, C., Kashnikov, Y., Renneke, J., Fursin, G.: Transforming GCC into a research-friendly environment - plugins for optimization tuning and reordering, function cloning and program instrumentation. In: 2nd Int'l Workshop on GCC Research Opportunities (2010)
18. Li, Y.-T., Malik, S.: Performance Analysis of Embedded Software using Implicit Path Enumeration. In: Workshop on Languages, Compilers, and Tools for Real-time Systems (1995)
19. Li, X., Roychoudhury, A., Mitra, T.: Modeling out-of-order processors for WCET analysis. Real-Time Systems Journal 34(3) (2006)
20. Lim, S.-S., Bae, Y.H., Jang, G.T., Rhee, B.-D., Min, S.L., Park, C.Y., Shin, H., Park, K., Kim, C.S.: An accurate worst case timing analysis technique for RISC processors. In: IEEE Real-Time Systems Symposium (1994)
21. Lim, S.-S., Kim, J., Min, S.-L.: A worst case timing analysis technique for optimized programs. In: International Conference on Real-Time Computing Systems and Applications (1998)
22. Metzlaff, S., Uhrig, S., Mische, J., Ungerer, T.: Predictable Dynamic Instruction Scratchpad for Simultaneous Multithreaded Processors. In: MEDEA Workshop (2008)
23. Paolieri, M., Quiones, E., Cazorla, F., Bernat, G., Valero, M.: Hardware Support for WCET Analysis of HRT Multicore Systems. In: Int'l Symposium on Computer Architecture (2009)
24. Puschner, P., Burns, A.: Writing temporally predictable code. In: 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (2002)
25. Ratsiambahotra, T., Cassé, H., Sainrat, P.: A Versatile Generator of Instruction Set Simulators and Disassemblers. In: Int'l Symp. on Performance Evaluation of Computer and Telecommunication Systems (2009)
26. Rochange, C., Bonenfant, A., Sainrat, P., Gerdes, M., Wolf, J., Ungerer, T., Petrov, Z., Mikulu, F.: WCET Analysis of a Parallel 3G Multigrid Solver Executed on the MERASA Multi-core. In: Int'l Workshop on Worst-Case Execution Time Analysis (2010)
27. Rochange, C., Sainrat, P.: A Context-Parameterized Model for Static Analysis of Execution Times. Transactions on High-Performance Embedded Architectures and Compilers 2(3) (2007)
28. Sen, R., Srikant, Y.N.: WCET estimation for executables in the presence of data caches. In: 7th International Conference on Embedded Software (2007)
29. Staschulat, J., Ernst, R.: Worst case timing analysis of input dependent data cache behavior. In: Euromicro Conference on Real-Time Systems (2006)
30. Theiling, H., Ferdinand, C.: Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In: IEEE Real-Time Systems Symposium (1998)
31. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström: The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools. ACM Transactions on Embedded Computing Systems 7(3) (2008)
32. Open source (Mixed-Integer) Linear Programming system, http://lpsolve.sourceforge.net/5.5/

Ubiquitous Verification of Ubiquitous Systems*

Reinhard Wilhelm and Matteo Maffei

Universität des Saarlandes, Saarbrücken, Germany
{wilhelm,maffei}@cs.uni-saarland.de

Abstract. Ubiquitous embedded computing systems expected to reliably perform one or more relevant tasks need design and verification methods currently not available. New envisioned applications and trends in system design increase this need. Several of these trends, e.g. function integration, concurrency, energy awareness, networking and their consequences for verification are considered in this article. It is described that, already in the past, verification was made possible only due to rules restricting the design and it is argued that even more so in the future the constructive influence on the design of hardware and software will be a necessary condition to keep the verification task tractable.

Keywords: Embedded systems, verification, security, networking, timing analysis, WCET, hard real-time, timing predictability.

1 Introduction

Ubiquitous embedded systems often combine an impressive number of requirements: They should be functionally correct as most other computing systems. They often need to react in real time. They should save energy if they are mobile, and they need to be secure if tampering is possible through an interface or a network. This article considers the verification challenge for such a combination of requirements. This challenge is quite formidable! Several of the individual properties are already very hard to verify. This is witnessed by undecidability and by complexity results for some of the tasks. An escape has always been to resort to simplified settings or to heuristic methods. However, these simplified problems were still hard enough. A modular approach to the challenge of verifying systems with such a combination of required properties seems to be the only solution. However, the interdependence between the different properties does not easily allow this. This is already clear from the example of timing validation where the interdependence of different architectural components introduces timing anomalies [35,43] and forces any sound timing analysis to analyze a huge architectural state space.

* The research reported herein was supported by the European Network of Excellence *ArtistDesign*, the Deutsche Forschungsgemeinschaft in SFB/TR 14 *AVACS*, the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement number 216008 (*Predator*), the initiative for excellence and the Emmy Noether program of the German federal government, and by Miur'07 Project SOFT (*Security Oriented Formal Techniques*)'.

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 47–58, 2010.

This article presents an analysis of past developments and of current trends with their implications for verification. It attempts to convey the message that *the trends in computer and software architecture and in system development increase the need for and the complexity of system verification*, and that *constructive influence on the design of hardware and software has to be exercised in order to keep the verification task tractable.*

The structure of this article is as follows. The development of the timing-analysis problem and the methods to solve it are presented in Section 2. It is shown how different notions of *state* evolved and how the state spaces to be analyzed exploded. Also, examples of how constructive influence was exercised to keep the problem tractable are given. In Sections 2.2 and 2.3, it is sketched how the introduction of concurrency and the need to save energy create new verification challenges. In Sections 2.4 and 3, two research directions are discussed in more detail. The first one is how to overcome difficulties in timing analysis, namely by making architectures timing predictable. The second concerns the security problems created by connecting embedded systems by networks. The state of the art in automatic verification is presented and open problems are given.

Several analogies between the two system properties, i.e., timing behavior and security, and the resulting verification problems are stressed:

- The security domain has seen correctness proofs of security protocols. However, they did not necessarily hold for their implementations since the proofs abstracted from essential system properties. Surprisingly, even proofs about the implementation on the source level may not be sufficient as witnessed in [44]. So, several levels have to considered for a total proof of security properties.

  Similarly, timing validation has been done on the specification level [16]. However, those prrofs don't carry over to the implementation level since they typically use unit-time abstraction, i.e., all transitions in the architecture take 1 unit of time. Neither can central parts of timing analysis be done on the source level since the source doesn't refer to the architecture.
- Compositionality of the resource behavior is the dream behind the AUTOSAR and IMA architecture movements in the automotive and avionics domains. So far, it has not been achieved.

  The security domain knows some compositionality results. However, they only hold under appropriate conditions.

2 Timing Analysis

Timing analysis of hard real-time systems is a good area to demonstrate how trends in application and system design have increased the pressure for sound verification methods and at roughly the same time the complexity of the verification task.

Figure 1 shows the development over time of the hardware and software architectures and the timing-analysis methods used in time-critical embedded systems. Each transition extended the state space to be explored, often adding one

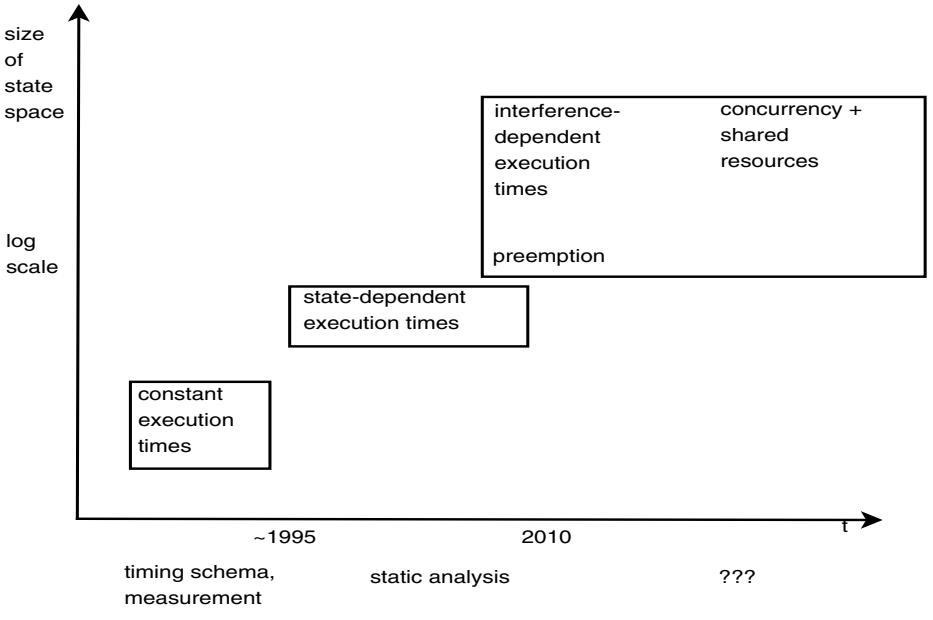


Fig. 1. Timing analysis — increasing the complexity of the architectures increases the complexity of the verification problems and makes established methods obsolete

more dimension and leading to an exponential growth of the state space. We will see in the course of the article that the development trends led to different notions of *state*.

Until the 90s, processor architectures used in time-critical embedded systems had instruction-execution times that were mostly independent of the execution state. The small onboard cache if any was often switched off. The processor handbook would give the (constant) number of cycles each instruction would take to execute. The most popular methods for timing analysis in these times were measurement und the so-called *timing schema* approach [46,39], sometimes in combination. Exhaustive measurement could be used if the input domain was sufficiently small. The state space to search was determined by the input domain. The transitions through the space of execution states of the architecture were abstracted away by the constant execution times. The timing-schema approach used induction over the structure of programs to compute upper timing bounds of programs from bounds of components.

2.1 State-Dependent Execution Times

EADS Airbus used the Motorola ColdFire processor in its A340 planes, a processor with a unified data/instruction cache of 8K size with a pseudo-round-robin replacement policy. This cache turned out to have very bad predictability properties [28]. Static-analysis based methods were still able to determine suffciently

precise bounds on the execution times [21]. The execution time of instructions depended strongly on the (architectural) state in which the instructions were executed. The state space to explore became the cartesian product of the data domain and the architectural state space. This was clearly too large to be exhaustively covered by measurement-based methods. Timing schemas could not easily be used any more since combining worst-case timings would produce too pessimistic results. Resorting to higher order schemas by adding a state parameter and combining timings with matching states avoided this pitfall [34].

It helped that task sets were expected to be scheduled non-preemptively. So, bounds on the execution time were determined for uninterrupted task execution. No interference from outside of the task needed to be considered. This was sufficient for many safety-critical avionics systems as they were typically synthesized from SCADE models [48]. We consider this as an example, where problem-aware developers of safety-critical systems avoided creating a problem that was not solvable by methods existing at the time.

At least in research, preemptive execution was considered [32,33]. However, it was clear that, for complexity reasons, the necessary determination of context-switch costs could not be performed at every program point. Delayed preemption was mostly assumed to keep the effort for the determination of program-point-specific context-switch costs tolerable. This, we consider as a second example for how constructive influence was taken to keep complexity within bounds.

Methods based on this work were improved, so that they are now finally mature enough to be interfaced to schedulability analyses [3].

Preemptive scheduling introduced a timing-analysis problem where the interference between tasks influenced the tasks' execution times. Another type of interference creates more severe problems with the advent of multi-processor/multi-core architectures and multi-threading software.

2.2 Concurrency

Future embedded system will be executed on manycore/multicore systems and will, thus, enjoy all performance, energy, and reliability advantages of these platforms. Several functions will be integrated on such platforms as witnessed in the AUTomotive Open System ARchitecture (AUTOSAR)[1] and the Integrated Modular Avioics (IMA) [49] examples.

However, this step also increases the verification pressure and increases the state space in another dimension by admitting potentially many different interleavings of the concurrent tasks' executions. Proving the functional correctness of concurrent systems is still a challenge. The IMA standard imposes *temporal* and *spatial partitioning* of the functions integrated on one execution platform. This eases the verification of functional correctness by avoiding the interference of tasks on a global state represented in the platforms memory hierarchy. This is another example of how a system-design discipline eases the verification task.

[1] See www.autosar.org

However, it does not prevent the interference on shared resources regarding their architectural state, i.e. their occupancy. It therefore does not support *compositionality* of the resource behavior. The performance of a function implemented on the platform may change when another function's implementation is replaced. This undermines the envisioned *incremental qualification. Resource isolation* of concurrently running tasks is one means to achieve this as will be later argued.

2.3 Power Awareness

Ubiquitous embedded systems whenever they are parts of mobile applications need to be designed in a power-aware fashion. Dynamic voltage scaling is often applied to reduce energy consumption [29]. Turning down the processor frequency has an impact on the performance and, thus, interacts with timing analysis. [45] describes an approach to timing analysis in combination with dynamic frequency scaling. However, the described method only works with simple processor architectures.

2.4 Predictability

One way to alleviate the timing-analysis task is to increase the predictability of the underlying architecture. This should be done on the single-core level [50] and, even more importantly, on the multi-core level [15]. The *predictability* notion has been around for a while [47], however, without a formal foundation.

Design for predictability of architectures is a very active area of current research. However, there is no agreement on the notion of predictability. The strictest notion requires execution-state independent timing behaviour of instructions. This direction is represented by the newly appeared XCore platform of XMOS[2] and by the PRET project [20]. Both do not use caches in their design as caches introduce a large variability of execution times. It is so far unclear how much performance has to be sacrificed.

The MERASA project concentrates on making architectural components more predictable. [37] describes architectural support for predictable symmetric multithreading. [38] presents a multi-core design enforcing a bounded delay on the access to a shared bus.

Our conception of the predictability of architectures admits architectural components introduced to increase average-case performance as long as the variability of their behaviour can be precisely and efficiently bounded by static analysis for a given software.

[42,41] have given precise notions for predictability of different cache replacement strategies (LRU, PLRU, FIFO, MRU). This work was the first to formally define cache predictability as *speed of recovery from uncertainty* and to rigorously compare different replacement policies. Similarly, the *sensitivity* to the

[2] https://www.xmos.com/products

initial state of cache performance for different replacement policies has been investigated. Cache performance under non-LRU policies are extremely sensitive to the initial cache state. Therefore, measurement-based approaches can yield results which are significantly lower than the actual WCET. The results led to the conclusion that LRU is superior to any other considered replacement policy and this under different criteria: performance, predictability, and therefore expected precision of cache analysis, and sensitivity. In addition, all approaches to determine the cache-related preemption delay only work for LRU.

[50] examines the relation between performance and analysis effort for the design of pipelines and buses and derived design guidelines.

[15] identifies *design rules for predictable multi-processors*. The first principles are to avoid interference on shared resources in the architecture and to allow the application designer to map applications to the target architecture without introducing new interferences that were not present in the application. This is because *interference by sharing* resources such as buses and caches is the main obstacle towards timing analysis for multi-core architectures. Thus, removal of sharing is the key to predictability. For costly and infrequently accessed resources such as I/O devices, interference costs can be bounded by imposing deterministic access protocols. This doesn't introduce much overestimation due to the generally low utilisation. Additional sharing (e.g., to meet cost constraints) is allowed if safe and sufficiently small delays for the access to shared resources can be guaranteed.

Compositionality. Compositionality of system properties is extremely important as it allows the component-wise verification of a system with a guarantee of the property for the whole system. Expectations towards compositionality of the predicted resource behavior, however, are misleading. Full compositionality can not be expected as the resources in embedded systems are bounded. All that can be expected is compositionality of the resource behavior given an unlimited supply. Hence, two proof obligations arise. Firstly, that one component's resource behaviour does not change that of another component assuming unbounded resources, and secondly, that the overall resource requirements are satisfiable. Incremental qualification faces the same limitation.

3 Networking and Security

3.1 Security Issues in Networked Embedded Systems

One of the distinctive features of modern embedded systems is the support for networking. This is motivated by the increasing demand of personalized services and collaborative platforms. Just to mention some examples, vehicular ad-hoc networks allow drivers and passengers to communicate with each other as well as with the roadside infrastructure in order to send and receive warnings about traffic jam, incidents, queues, and so on; some household thermostats offer Internet connectivity to let the owner switch on the heating a certain time before the arrival; some hospitals use wireless networks for patient care equipment.

Although networking paves the way for the development of services that were not imaginable a few years ago, it also poses serious security issues. For instance, even the notoriously quoted refrigerator who, connected to the internet to order food and drinks, poses such a risk. Who would be happy to find all of ALDI's beer supply infront of one's door when a hacker enjoyed playing a nice joke? More seriously, security and trust mechanisms have to be introduced in vehicular ad-hoc networks in order to protect the privacy of drivers and to prevent malicious drivers or corrupted devices from broadcasting false warnings [40]; similarly, access control rules have to be enforced in order to ensure that only the house owner can control the thermostat therein [30]; perhaps more surprisingly, it has recently been shown that several attacks can be mounted on implantable cardioverter defibrillators, compromising patient safety and patient privacy, up to inducing electrical shocks into the patient's heart [27]! These attacks have been discovered by applying reverse engineering and eavesdropping techniques to a previously unknown radio communication protocol.

From this perspective, the recent trend to introduce wired and wireless networks on planes and to rely on software solutions to ensure the intended safety and security guarantees further witnesses the dramatic need of automated verification tools for the security of networked embedded systems. A FAA (Federal Aviation Administration) document dating back to 2008 points out weaknesses in the communication network within the Boeing's new 787 Dreamliner. The proposed architecture of the 787," the FAA stated, "allows new kinds of passenger connectivity to previously isolated data networks connected to systems that perform functions required for the safe operation of the airplane...The proposed data network design and integration may result in security vulnerabilities from intentional or unintentional corruption of data and systems critical to the safety and maintenance of the airplane" [2].

3.2 Automated Verification of Cryptographic Protocols

The security model that should be taken into account in the analysis of networked embedded systems comprises internal attackers (i.e., malicious users and corrupted devices) as well as external attackers. The security desiderata are application dependent and may include the secrecy and integrity of data, access control policies, trust policies, and user anonymity.

Cryptographic protocols constitute core building blocks for designing systems that stay secure even in the presence of malicious entities. The design of security protocols has long been known to be a challenging task, which is even more challenging in the context of embedded systems since the limited amount of available resources often rules out the possibility to employ expensive, powerful cryptographic operations (e.g., zero-knowledge proofs and secure multiparty computations). Even in heavily simplified models where the complexity of cryptographic operations is abstracted away and cryptographic messages are modelled as symbolic terms, security properties of cryptographic protocols are in general undecidable. Additionally, security analyses of such protocols are

awkward to make for humans, due to the complexity induced by multiple interleaved protocol runs and the unpredictability of the attacker behavior. Formal methods, and in particular static analysis techniques such as type systems [1,24,14,5,7,9], abstract interpretation [23,12,13,6], and theorem proving [11] proved to constitute salient tools for reliably analyzing security protocols. Nowadays, the analysis of sophisticated properties, such as anonymity, privacy, and access control policies, is within the scope of automated verification tools and the running time for such analyses ranges from a few seconds to a couple of hours, depending on the complexity of the protocol and of the cryptographic primitives.

3.3 Towards a Security Analysis of Embedded Systems

Despite these promising results, however, the verification of security properties of networked embedded systems is still an open issue. The main reason is that the aforementioned automated analysis techniques focus on the logic of the protocol (i.e., the way cryptographic messages are exchanged) and tend to abstract away from its implementation. Consequently, a gap often exists between the verified protocol models and the actually deployed implementations. Hence, even if the abstract model is proved to be safe, security flaws may still affect its implementation [10]. Only recently, some works have tackled the analysis of the source code of protocol implementations, with a specific focus on functional languages [10,9] and C code [25,17]. Still the semantics of the programming languages is idealized (e.g., by encoding in the lambda calculus) and the verified models abstract a number of potentially troublesome details. Recent papers have touched the security analysis of bytecode, hardware language, and timing behavior, but they mainly focused on information flow properties and non-concurrent code [31,22,8]. The verification of distributed implementations of security protocols is still an open issue and it has been recognized as the grand challenge of the next twenty years at the 21st Computer Security Foundation Symposium (CSF'08).

The security dimension of computing systems is not independent of the other dimensions listed so far. The secrecy of data, in particular, is a "local" property crucially depending on implementation and hardware details. The observation of the timing behavior and of the energy consumption of program parts, for instance, can offer covert channels to leak private data [51]. Higher-level security properties (e.g., authentication, user anonymity, and distributed access control) are however typically "global": they build on top of the secrecy of some data, such as keys, passwords, and credentials, but for the rest they solely depend on the messages exchanged on the network. An interesting research direction that is worth to be explored, consequently, is the automated verification of secrecy on detailed hardware models and the investigation of compositionality results ensuring that global security properties that are verified on abstract models carry over to the actual implementation, as long as this locally preserves the expected secrecy properties and complies with the intended communication protocol.

3.4 Design Guidelines to Simplify the Analysis

A careful architecture design may strengthen the security of the system and help to reduce the complexity of the analysis. Here we discuss two important aspects, namely, *hardware solutions* and *compositionality principles*.

Hardware solutions, in conjunction with software-based security mechanisms, enhance the performance and ensure the correctness of basic cryptographic operations, thus improving the security of the system and reducing the complexity of the analysis. By way of example, the secrecy of sensible data, which as discussed above constitutes the building block of higher level security properties, can be enforced by using dedicated cryptographic chips, such as the trusted platform modules (TPMs). These cryptographic chips facilitate the secure generation of cryptographic keys, allow for securely storing these keys, and offer support for the efficient implementation of a number of cryptographic operations, including advanced schemes such as zero-knowledge proofs. Nowadays, TPMs are included in most high level laptops and their applications include secure disk encryption, password protection, and digital right management.

Security protocols in general do not enjoy compositionality properties: If we consider two protocols that are secure when executed in physically separated systems, we are not guaranteed that their concurrent execution in the same environment achieves the same security properties. Nevertheless, there exist design principles that can be followed to obtain compositionality guarantees. These principles include the usage of distinct cryptographic keys or disjoint encryption schemes [26], the tagging of cryptographic messages [36,18], and certain patterns enforced by compositional analysis techniques [14,19,4]. Protocol compositionality is a crucial aspect in the analysis of cryptographic protocols as it allows for the independent verification of each single component, thus significantly reducing the state space, yet obtaining in the end security guarantees for the system as a whole.

4 Conclusions

We have discussed several required properties of ubiquitous embedded systems and the resulting verification problem, highlighting open issues and suggesting directions of future research. Security and timing requirements were discussed in more depth. It was described how design methods and compositionality properties have helped to keep the verification problem tractable and argued that this will also hold in the future.

References

1. Abadi, M.: Secrecy by typing in security protocols. Journal of the ACM 46(5), 749–786 (1999)
2. Administration, F.A.: Federal Register, vol. 73(1) (January 2, 2008)

3. Altmeyer, S., Maiza, C., Reineke, J.: Resilience analysis: tightening the CRPD bound for set-associative caches. In: Lee, J., Childers, B.R. (eds.) LCTES, pp. 153–162. ACM, New York (2010)
4. Andova, S., Cremers, C., Gjøsteen, K., Mauw, S., Mjølsnes, S.F., Radomirović, S.: A framework for compositional verification of security protocols. Information and Computation 206(2-4), 425–459 (2008)
5. Backes, M., Cortesi, A., Focardi, R., Maffei, M.: A calculus of challenges and responses. In: Proc. 5rd ACM Workshop on Formal Methods in Security Engineering (FMSE), pp. 101–116. ACM Press, New York (2007)
6. Backes, M., Cortesi, A., Maffei, M.: Causality-based abstraction of multiplicity in cryptographic protocols. In: Proc. 20th IEEE Symposium on Computer Security Foundations (CSF), pp. 355–369. IEEE Computer Society Press, Los Alamitos (2007)
7. Backes, M., Hriţcu, C., Maffei, M.: Type-checking zero-knowledge. In: 15th Proc. ACM Conference on Computer and Communications Security, pp. 357–370. ACM Press, New York (2008)
8. Barthe, G., Pichardie, D., Rezk, T.: A certified lightweight non-interference java bytecode verifier. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 125–140. Springer, Heidelberg (2007)
9. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. In: CSF '08: Proceedings of the 2009 21st IEEE Computer Security Foundations Symposium, pp. 17–32. IEEE Computer Society, Los Alamitos (2008)
10. Bhargavan, K., Fournet, C., Gordon, A.D., Tse, S.: Verified interoperable implementations of security protocols. In: Proc. 19th IEEE Computer Security Foundations Workshop (CSFW), pp. 139–152. IEEE Computer Society Press, Los Alamitos (2006)
11. Blanchet, B.: An efficient cryptographic protocol verifier based on Prolog rules. In: Proc. 14th IEEE Computer Security Foundations Workshop (CSFW), pp. 82–96. IEEE Computer Society Press, Los Alamitos (2001)
12. Bodei, C., Buchholtz, M., Degano, P., Nielson, F., Nielson, H.R.: Static validation of security protocols. Journal of Computer Security 13(3), 347–390 (2005)
13. Boichut, Y., Genet, T.: Feasible trace reconstruction for rewriting approximations. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 123–135. Springer, Heidelberg (2006)
14. Bugliesi, M., Focardi, R., Maffei, M.: Dynamic types for authentication. Journal of Computer Security 15(6), 563–617 (2007)
15. Burguiere, C., Grund, D., Reineke, J., Wilhelm, R., Cullmann, C., Ferdinand, C., Gebhard, G., Triquet, B.: Predictability considerations in the design of multi-core embedded systems. In: Embedded Real Time Software and Systems, ERTSS (2010)
16. Campos, S.V.A., Vale, S., Campos, A., Gerais, M., Horizonte, B., Clarke, E.: Analysis and verification of real-time systems using quantitative symbolic algorithms. Journal of Software Tools for Technology Transfer 2, 260–269 (1999)
17. Chaki, S., Datta, A.: Aspier: An automated framework for verifying security protocol implementations. In: CSF '09: Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium, pp. 172–185. IEEE Computer Society, Los Alamitos (2009)
18. Cortier, V., Delaune, S.: Safely composing security protocols. Formal Methods in System and Design 34(1), 1–36 (2009)
19. Datta, A., Derek, A., Mitchell, J., Roy, A.: Protocol composition logic (pcl). Electronic Notes on Theoretical Computer Science 172, 311–358 (2007)

20. Edwards, S.A., Lee, E.A.: The case for the precision timed (pret) machine. In: DAC, pp. 264–265. IEEE, Los Alamitos (2007)
21. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: WCET Determination for a Real-Life Processor. In: Henzinger, T., Kirsch, C. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 469–485. Springer, Heidelberg (2001)
22. Genaim, S., Spoto, F.: Information flow analysis for java bytecode. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 346–362. Springer, Heidelberg (2005)
23. Genet, T., Tong, V.: Reachability analysis of term rewriting systems with timbuk. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS (LNAI), vol. 2250, pp. 695–706. Springer, Heidelberg (2001)
24. Gordon, A.D., Jeffrey, A.: Types and effects for asymmetric cryptographic protocols. Journal of Computer Security 12(3), 435–484 (2004)
25. Goubault-Larrecq, J., Parrennes, F.: Cryptographic protocol analysis on real C code. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 363–379. Springer, Heidelberg (2005)
26. Guttman, J.D., Thayer, F.J.: Protocol independence through disjoint encryption. In: Proc. 13th IEEE Computer Security Foundations Workshop (CSFW), pp. 24–34. IEEE Computer Society Press, Los Alamitos (2000)
27. Halperin, D., Heydt-Benjamin, T.S., Ransford, B., Clark, S.S., Defend, B., Morgan, W., Fu, K., Kohno, T., Maisel, W.H.: Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In: SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy, Washington, DC, USA, pp. 129–142. IEEE Computer Society, Los Alamitos (2008)
28. Heckmann, R., Langenbach, M., Thesing, S., Wilhelm, R.: The Influence of Processor Architecture on the Design and the Results of WCET Tools. IEEE Proceedings on Real-Time Systems 91(7), 1038–1054 (2003)
29. Kim, W., Kim, J., Min, S.: A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis. In: DATE '02: Proceedings of the Conference on Design, Automation and Test in Europe, Washington, DC, USA, p. 788. IEEE Computer Society, Los Alamitos (2002)
30. Koopman, P.: Embedded system security. Computer 37(7), 95–97 (2004)
31. Köpf, B., Basin, D.A.: An information-theoretic model for adaptive side-channel attacks. In: Proc. 14th ACM Conference on Computer and Communications Security, pp. 286–296 (2007)
32. Lee, C., Han, J., Seo, Y., Min, S., Ha, R., Hong, S., Park, C., Lee, M., Kim, C.: Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling. In: Proceedings of the IEEE Real-Time Systems Symposium (December 1996)
33. Lee, S., Lee, C.-G., Lee, M.S., Min, S.L., Kim, C.S.: Limited Preemptible Scheduling to Embrace Cache Memory in Real-Time Systems. In: Müller, F., Bestavros, A. (eds.) LCTES 1998. LNCS, vol. 1474, pp. 51–64. Springer, Heidelberg (1998)
34. Lim, S.-S., Bae, Y.H., Jang, G.T., Rhee, B.-D., Min, S.L., Park, C.Y., Shin, H., Park, K., Moon, S.-M., Kim, C.S.: An accurate worst case timing analysis for RISC processors. IEEE Transactions on Software Engineering 21(7), 593–604 (1995)
35. Lundquist, T., Stenström, P.: Timing Anomalies in Dynamically Scheduled Microprocessors. In: 20th IEEE Real-Time Systems Symposium (1999)
36. Maffei, M.: Tags for multi-protocol authentication. In: Proc. 2nd International Workshop on Security Issues in Coordination Models, Languages, and Systems (SECCO '04). Electronic Notes on Theoretical Computer Science, pp. 55–63. Elsevier Science Publishers Ltd., Amsterdam (2004)

37. Mische, J., Guliashvili, I., Uhrig, S., Ungerer, T.: How to enhance a superscalar processor to provide hard real-time capable in-order smt. In: Müller-Schloer, C., Karl, W., Yehia, S. (eds.) ARCS 2010. LNCS, vol. 5974, pp. 2–14. Springer, Heidelberg (2010)
38. Paolieri, M., Quiones, E., Cazorla, F.J., Bernat, G., Valero, M.: Hardware support for wcet analysis of hrt multicore systems. In: The 36th International Symposium on Computer Architecture, ISCA 2009 (2009)
39. Puschner, P., Koza, C.: Calculating the maximum execution time of real-time programs. Real-Time Systems 1, 159–176 (1989)
40. Raya, M., Hubaux, J.-P.: The security of vehicular ad hoc networks. In: SASN '05: Proceedings of the 3rd ACM Workshop on Security of Ad hoc and Sensor Networks, pp. 11–21. ACM, New York (2005)
41. Reineke, J.: Caches in WCET Analysis. PhD thesis, Universität des Saarlandes, Saarbrücken (2008)
42. Reineke, J., Grund, D., Berg, C., Wilhelm, R.: Timing Predictability of Cache Replacement Policies. Real-Time Systems 37(2), 99–122 (2007)
43. Reineke, J., Wachter, B., Thesing, S., Wilhelm, R., Polian, I., Eisinger, J., Becker, B.: A definition and classification of timing anomalies. In: Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis (July 2006)
44. Reps, T.W., Balakrishnan, G.: Improved memory-access analysis for x86 executables. In: Hendren, L.J. (ed.) CC 2008. LNCS, vol. 4959, pp. 16–35. Springer, Heidelberg (2008)
45. Seth, K., Anantaraman, A., Mueller, F., Rotenberg, E.: Fast: Frequency-aware static timing analysis. ACM Trans. Embed. Comput. Syst. 5(1), 200–224 (2006)
46. Shaw, A.C.: Reasoning About Time in Higher-Level Language Software. IEEE Transactions on Software Engineering 15(7), 875–889 (1989)
47. Stankovic, J.A., Ramamritham, K.: Editorial: What is predictability for real-time systems? Real-Time Systems 2(4), 247–254 (1990)
48. Thesing, S., Souyris, J., Heckmann, R., Randimbivololona, F., Langenbach, M., Wilhelm, R., Ferdinand, C.: An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software Systems. In: Proceedings of the Performance and Dependability Symposium, San Francisco, CA (June 2003)
49. Watkins, C.B., Walter, R.: Transitioning from federated avionics architectures to integrated modular avionics. In: 26th Digital Avionics Systems Conference DASC (2007)
50. Wilhelm, R., Grund, D., Reineke, J., Pister, M., Schlickling, M., Ferdinand, C.: Memory hierarchies, pipelines, and buses for future time-critical embedded architectures. IEEE TCAD 28(7), 966–978 (2009)
51. Wray, J.C.: An analysis of covert timing channels. In: IEEE Symposium on Security and Privacy, p. 2 (1991)

A Model–Based Design Methodology with Contracts to Enhance the Development Process of Safety–Critical Systems

Andreas Baumgart[1], Philipp Reinkemeier[1], Achim Rettberg[2], Ingo Stierand[2], Eike Thaden[1], and Raphael Weber[1]

[1] OFFIS, Escherweg 2, 26121 Oldenburg, Germany
[2] Carl von Ossietzky University Oldenburg, OFFIS, Escherweg 2, 26121 Oldenburg, Germany

Abstract. In this paper a new methodology to support the development process of safety–critical systems with contracts is described. The meta–model of Heterogeneous Rich Component (HRC) is extended to a Common System Meta–Model (CSM) that benefits from the semantic foundation of HRC and provides analysis techniques such as compatibility checks or refinement analyses. The idea of viewpoints, perspectives, and abstraction levels is discussed in detail to point out how the CSM supports separation of concerns. An example is presented to detail the transition concepts between models. From the example we conclude that our approach proves valuable and supports the development process.

1 Introduction

In many application domains the field of distributed embedded systems has an increasing impact on the development of products. Since the products are often *safety critical systems* where erroneous behavior may lead to hazardous situations, standards such as DO–178B [12] (avionics) and ISO 26262 [6] (automotive) describe development processes in order to ensure functional correctness and safety properties. Only if a product is developed according to the relevant standards the respective certification authorities will approve the final product.

Another arising issue is the increasing size and complexity of the software (and hardware) of such systems. There are a number of reasons for this: The implementation of new functions often leads to inconsistent systems. Other functions have to be modified and in quite a few cases a modified technical architecture might be necessary to compensate the overall impact of new functions. It is a challenge to manage the development of such complex systems. A crucial part is a continuous and strong methodology to aid the development process in multiple dimensions.

In order to deal with complexity, existing design methodologies provide abstraction and refinement techniques. Abstraction allows the designer to concentrate on the essentials of a problem, and to enable concurrent development. Another important feature is the support for re–use. With this, systems can

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 59–70, 2010.

evolutionary be constructed saving time and costs. Many methodologies provide re–use by a concept of components. Components encapsulate logical units of behavior that can be instantiated in different contexts. This also enables an incremental development process by a concept of refinement for components.

While many existing meta–models partially support such methodologies, they often do not cover the whole design flow from initial requirements down to a final implementation and do not support traceability of that process. A meta–model, to our understanding, provides the designer with the necessary modeling entities to comprehensively compose a system. In order to derive verifyable and executable embedded system specifications a meta–model needs to be interpreted which can be done by an underlying semantics. A methodology thereby describes the design steps of how a system should be modeled utilizing a meta–model.

In this paper we present a meta–model currently under development in the projects SPES2020 [16] and CESAR [14]. Some underlying concepts were already outlined in [17] and will be further detailed in this work. In the following, we will refer to our meta–model as the ***C**ommon **S**ystems **M**eta–model* (CSM). CSM features concepts to support component–based design, to specify formal and non–formal requirements and to link them to components. It supports the seamless design flow from the initial requirement specification down to the implementation. The meta–model for *Heterogeneous Rich Components* (HRC), developed in the SPEEDS [15] project, has been extended and constitutes the foundation of CSM. Thus, it benefits from the semantic foundation of HRC and provides analysis techniques such as compatibility checks or refinement analysis.

There already exist many meta–models, some of which will be briefly outlined and compared to CSM in Section 2. Since HRC is the core of our meta–model, we will give a short introduction to HRC in Section 3. Section 4 details how the CSM generically divides models to master complexity. In Section 5 we describe important CSM concepts along with an example model. The last section will draw a conclusion and give an outlook.

2 Architecture Description Languages

Many well established meta–models address a component–based development process including composition concepts. Some of them focus on a particular application domain and/or a certain stage of the development process. In the following we will describe the most relevant meta–models.

EAST–ADL [1] for instance aims at providing an open integration platform for modeling automotive systems and concentrates on early phases of development, i. e. specifying features of the product and analyzing its functions. However, modeling concepts for behavioral specification of functions are not part of EAST–ADL. According to the EAST–ADL specification UML state machines can be used here, but the semantics of their integration is not well defined. The CSM does not support the modeling of behavior directly. This is achieved through the concept of HRC contracts inherited from SPEEDS project to specify requirements for functional and non–functional behavior of components in conjunction with assumptions on their environments. In EAST–ADL requirements

can be categorized (e. g. different ASIL–levels) and also be linked to components — but they have no underlying formal semantics. The idea of having predefined abstraction levels in EAST–ADL is a valuable concept to separate models with different concerns, i. e. a functional model and an architectural design model. Thus, in CSM we adopt this concept in a more general way, which we call *perspectives*.

AUTOSAR [3] is based on a meta–model also targeting the automotive domain, but is utilized in a later step in the development process as it is more concerned with software configuration and integration aspects. The provided concepts for software component specification are nearly agnostic to the actual functionality a software component realizes. Instead they concentrate on defining modeling artifacts coupled with rules for code–generation to gain properly defined interfaces in order to ease the integration task. Concepts supporting functional modeling or earlier stages of the development process like abstraction levels are intentionally missing. Additionally, AUTOSAR does not consider requirement specifications, except for the new *Timing Extensions* [2]. In contrast to AUTOSAR CSM is meant to support a comprehensive modeling process starting from early requirements and going down to the actual implementation in a domain–independent way. If used in an automotive context, CSM models can be used to generate AUTOSAR artifacts which then can be used in native AUTOSAR configuration tools.

The AADL [5] is a modeling language that supports modeling and early and repeated analyses of a system's architecture with respect to performance–critical properties through an extendable notation (annexes). Furthermore, a tool framework and a formal semantics improves the usability and the usefulness in conjunction with analysis tools. AADL supports modelling of soft- and hardware components and their interfaces. These components serve as means to define the structure of an embedded system. Other features include functional interfaces like data IOs and non–functional aspects like timing and performance properties. The combination of components (connecting data in- and outputs or deployment of software on hardware) can be defined and modeled. Each of these components can contain a group connectors between interfaces of components forwarding control- or data–flows. Another feature of the AADL are so called modes attached to components. They represent alternative configurations of the implementation of a component. Like in EAST–ADL there is a way to describe system components, their interfaces, and the data they interchange. However, the designer starts to work at a stage where the software and hardware components are already explicitly identified and separated, i. e. modeling on higher levels of abstraction is not possible. Special annex libraries would have to be defined for exploiting the full potential of AADL, i. e. a formal specification of requirements and behavior not only limited to real–time specific constraints. Modeling on different levels of abstraction with refinement relations between abstract and more concrete artifacts is not supported in the AADL but might be useful to address e. g. separation of concerns or process–specific design steps.

The Systems Modeling Language (SysML) [9] standardized by the Object Management Group (OMG) extends UML2 with concepts for the development of embedded systems and is not tailored to a specific application–domain like EAST–ADL. Hardware, software, information, personnel, and facility aspects can be addressed. SysML can be used for the general notation of such a system so there is no explicit semantics for all elements and relationships. Furthermore, the language does not define architecture levels but provides means to describe them. There are structural block diagrams for component–based modeling, behavioral descriptions, requirements, and use cases. Moreover, SysML provides trace links which can be used to relate elements of different architectural levels: An element of an architectural level can realize another element, and an architectural property can be allocated to a property of another architecture model. Many concepts of SysML have been adopted in CSM, but key concepts like the concept of contracts with a rigid formal semantics are missing.

The *UML profile for Modeling and Analysis of Real-Time Embedded Systems* (MARTE) [10] adds capabilities for the real–time analysis of embedded systems to UML. It consists of several subprofiles, one of which providing a general concept of components. A subprofile called *NFPs* provides means to attach non–functional properties and constraints to design artifacts, that would later be subject to a real–time analysis. While this profile and its meta–model is agnostic to the application–domain, it does not detail the way constraints are specified. Despite being modular and open to other viewpoints, MARTE does not have a common underlying semantics that can be utilized for all viewpoints. In contrast, the HRC foundation of CSM has an automata–based semantics that can also be used for other viewpoints like safety and thus enables the specification of cross–viewpoint dependencies. This HRC foundation will be described in more detail in the next section.

3 Heterogenous Rich Components

HRC (*Heterogeneous Rich Component*), which originates from the european SPEEDS project, constitutes the core of the newly proposed meta–model. This section gives a short introduction to the concepts of HRC.

3.1 Structure

The meta–model provides basic constructs needed to model systems like components with one or more ports and connections (bindings) between them. A port aggregates multiple interaction points of the component typed by interfaces. These interfaces can be either flows that type data–oriented interaction points or services which type service–oriented interaction points. The interaction points of HRCs can be connected by means of bindings: Flows to flows and services to services. Either interconnections between ports and all their aggregated interaction points can be established or a subset of these interaction points are bound.

Components may have an inner structure consisting of subcomponents, their bindings between each other and their bindings from/to ports of their owning component. In the latter case the bindings specify a delegation of the flows or services to the inner component. This approach to model structure also common with most other meta–models for component based design, provides great reuse–capabilities and supports decomposition. Figure 1 depicts the concepts for modeling structure and their relations.

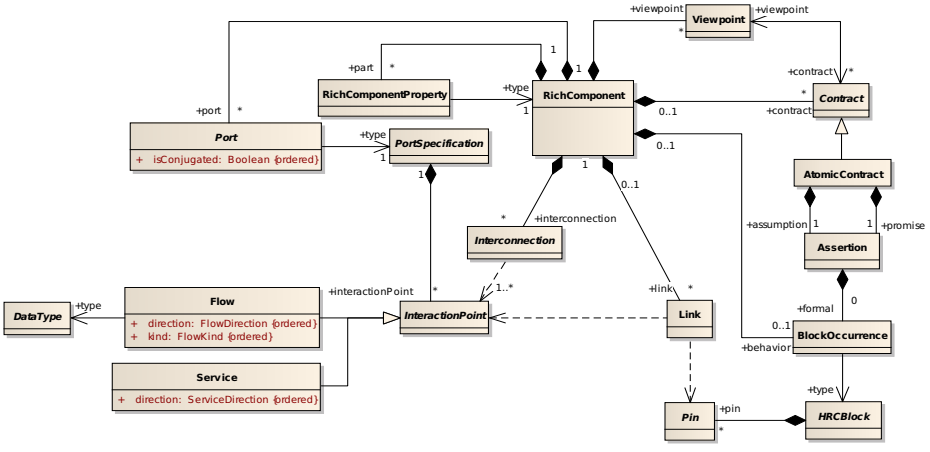


Fig. 1. HRC meta–model cut–out

3.2 Behavior

The dynamics of an HRC can be specified by HRC state machines, which are hybrid automata with a C-like action language. These automata are wrapped by so called *HRCBlocks*, that expose parts of the automata on their *pins* to interact with them. In turn these pins can be linked to interaction points of the owning HRC, thus exposing the dynamics of the component.

The concept of HRCBlocks is used in different contexts, where a dynamic specification is needed. First the implementation of a component can be specified as an HRCBlock, consisting of an HRC state machine. More important HRCBlocks can be used when specifying the requirements of a component by so called contracts. These concepts for specifying behavior and contracts are depicted on the right hand side of figure 1. While the component–port–interface concepts, inherent to many other metamodels, allow to specify a static contract for a component, they often do not account for the dynamics of that interface. One of the key concepts in HRC is the ability to abstract from the actual implementation of components and to specify the required behavior using *contracts*. The idea of contracts is insprired by Bertrand Meyer's programming language Eiffel and its *design by contract* paradigm [8]. In HRC contracts are a pair consisting of an assumption and a promise, both of which are specified by an HRC block. An

assumption specifies how the context of the component, i. e. the environment from the point of view of the component, should behave. Only if the assumption is fulfilled, the component will behave as promised. This enables the verification of virtual system–integration at an early stage in a design flow, even when there is no implementation yet. The system decomposition during the design process can be verified with respect to contracts. Details about the semantics of HRC are given in [11], and [7] describes what a design process utilizing HRC would look like by means of an example.

3.3 Viewpoints

HRC allows to group one or more contracts together and assign them to user–defined viewpoints. Examples for viewpoints are realtime, safety, performance, or power consumption. While contracts associated with different viewpoints are based on the same underlying semantics, viewpoints support the developer to separate different concerns.

4 Modeling along the Development Process

When developing an embedded system an architecture is regarded in different *perspectives* at several *abstraction levels* during the design process as mentioned before. Figure 2 illustrates a generalized V–model which has become a well established model for many development processes. The V–model shows how an embedded system can be developed along several abstraction levels starting with user requirements for operational scenarios, analysis of functional blocks with refined requirements, design decisions with derived requirements and a final implementation. The developed product has to be tested on all levels so the implementation integration is tested. The behavior of the system design is verified, the functionality is validated and the product is evaluated during its operation. On each level the product architecture is regarded in different perspectives. So on an operational level e. g. non–functional features and use cases, on functional analysis level a perspective with functional blocks and on design level logics, software, hardware and geometry can be considered. Models on each perspective reflect different viewpoints. A viewpoint “Safety” might be regarded in every perspective but a viewpoint “Realtime” is not regarded in a geometric perspective and viewpoint “Cost” is not regarded when considering operational use cases.

4.1 Abstraction Levels

In CSM we introduce the generic concept of abstraction levels. In a model there can be several concrete abstraction levels to represent the different levels of granularity. We furthermore define an ordering relation between abstraction levels where the highest abstraction level has the coarsest granularity in terms of description of the components. Lower abstraction levels are allowed to rearrange

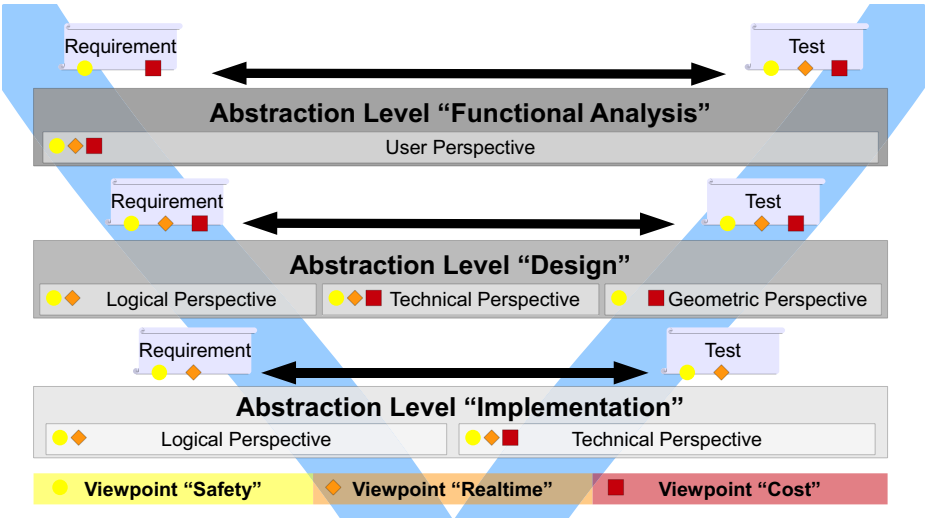


Fig. 2. Process model with different abstraction levels, perspectives and viewpoints

the functionality that is specified in a next higher abstraction level into different sets of components with respect to certain well–formedness properties. While the modeling granularity on lower abstraction levels is finer, the components still have to respect the functional and non–functional properties defined for their higher–level counterparts. This relation is formally defined in HRC based on HRC contracts as dominance relation (term used in SPEEDS) or entailment relation (renamed concept). Details for this can be found in Section 5.4.

For the CSM we observed that design processes are very diverse for different domains like automotive and avionics. A fixed set of predefined abstraction levels is not sufficient to handle the differences of multiple application domains. For example, in terms of CSM the EAST–ADL abstraction levels are just one possible instanciation of our generic abstraction level concept. In other application domains e. g. avionics or even for different companies in the same domain, the set of actual used concrete abstraction levels can be tailored to the specific needs. Each abstraction level contains one ore more perspectives which will be explained in the following.

4.2 Perspectives

Apart from the distinction of different user–defined abstraction levels, an abstraction level itself contains a set of model representations that reflect different aspects of the whole product architecture on the respective level of abstraction. We call such a distinct model representation on one abstraction level a “Perspective”. Such a perspective can contain a model representation containing hierarchical elements and element interactions. Currently we distinguish

four perspectives on each abstraction level: A user perspective which contains function–centric models, requirements engineering models, etc.; a logical perspective which describes the logical structure of the system with components, ports, connections, contracts, etc.; a technical perspective that focusses on physical hardware systems including communication buses; and a geometric model of the product.

The generic concept of different perspectives in abstraction levels is derived from the partitions of EAST–ADL abstraction levels and from an analysis of architecture partitioning. Taking a look at EAST–ADL one can see that one distinct EAST–ADL architecture level a parted into different perspective models: The design architecture level contains a functional as well as a hardware perspective. In [4] three architectures are defined namely a user level, a logical architecture and a technical architecture which are perspectives that can be found on several abstraction levels with a certain level of granularity.

On each perspective different viewpoints such as “Safety”, “Realtime” or “Cost” are regarded which will not be further discussed in this paper. Combining all perspectives of one abstraction level provides a model description of the whole architecture at one level of granularity. Element interfaces of different perspectives do not need to be compatible to each other. However, elements of one perspective can be allocated to elements of another perspective e. g. a feature is allocated to a function and a function is allocated to a hardware element and so on. In a graphical notation language such as SysML such an allocation is denoted by an “allocate” abstraction link.

The palette of actually used perspectives and elements can be different at each abstraction level e. g. distinction between hardware and software is not regarded in one abstraction level but in another. Thus, when descending to a lower level of abstraction elements of one perspective may be realized by elements in multiple finer–grained models of different perspectives.

5 A New Meta–model with Example

This section will cover the most important features of our meta–model CSM starting from a top abstraction level where there is no hard- or software but only requirements and a basic idea, what the final product should be able to do. During the design process components are conceptualized to iteratively support the deduction of solutions from requirements. Further down on the lower levels of abstraction more concrete components can be devided into functions mapped to resources (e. g. software running on hardware). The descriptions will mainly focus on the support of the meta-model for development-processes involving different abstraction levels and perspectives wrt. the system under development. We will illustrate the usage of the concepts by a running example (the wheel braking system of an aircraft), that has been inspired by an example from the standard ARP4761 [13].

5.1 From Required Features to Components

When starting the design of an embedded system one usually has consider a) the desired functionality of the system, b) the dedicated environment wherein the systems should work, and c) the existing regulations for building such systems. From this information the initial requirements concerning the system to design can be derived. As an example, a subset of required features concerning an aircraft's wheel braking system is displayed in Figure 3.

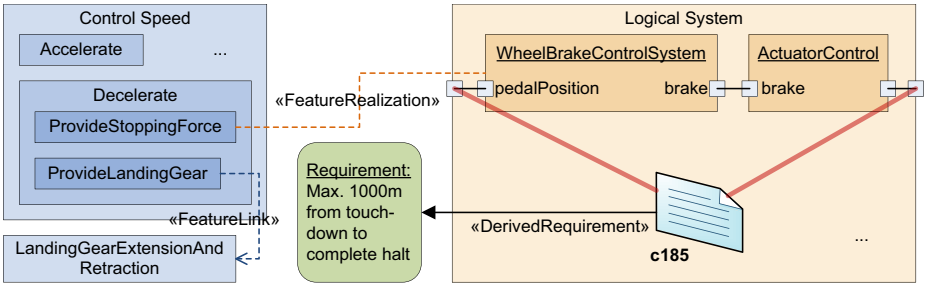


Fig. 3. Top abstraction level: Feature decomposition and initial logic architecture

After the initial requirements specification via features a first high level logic component decomposition is done. In our example the aircraft to design has (among other components) a wheel braking control system and an actuator controller. To trace from where these components originate one can annotate feature realization connections to represent which feature is realized through which component. Required features can also depend on other features, this is displayed with a feature link. In our example there is a requirement which informally states that the aircraft shall stop within 1000 meters after touchdown. On the right side, in the logical system, this requirement was refined into multiple other requirements, one of which appears as the formally specified contract *c185*. This contract, for instance, requires the system to react to a pilot pushing the brake pedal with a signal towards the actuator control within a certain time–frame (*Assumption:* "PedalPosition is available && no system error" *Promise:* "Delay from *pedalPosition.e* to *brake.e* within [10,18] time unit [ms]").

5.2 Decomposing the System: Towards Lower Abstraction Levels

So far we presented a high level view on the system under development, usually referred to as System Model. In Figure 4 the logical perspective on the system abstraction level is shown on top. Below that the logical perspective of the next lower abstraction level can be seen. We used a one–on–one realization relation between components on the different levels. The contract *C185* used on the higher level is connected to a derived contract *C185.13* on the lower level using an *Entailment–relation*. This relation states that both contracts are not necessarily

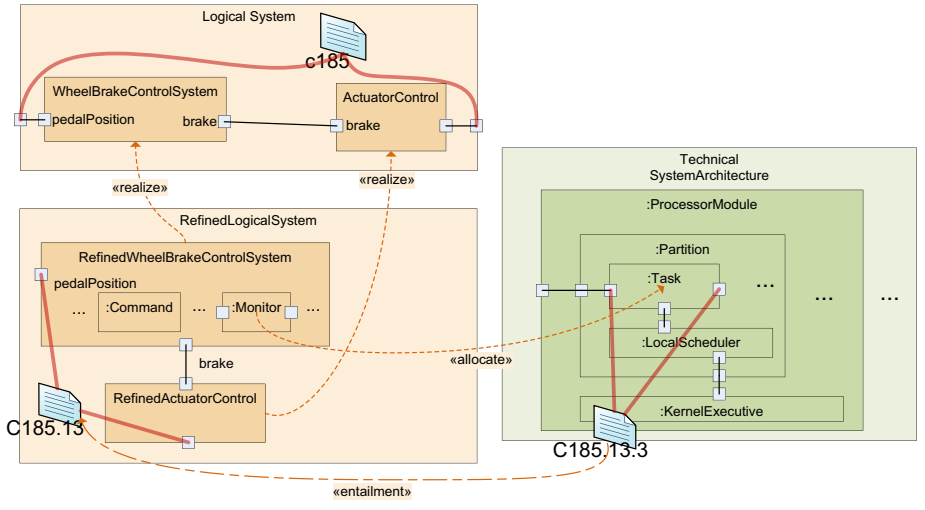


Fig. 4. System refinement and function allocation

identical, but the contract on the lower level is a refinement of its pendant on the higher level according to the SPEEDS semantics [11]. A new perspective is introduced on the lower abstraction layer: The *Technical Perspective*.

5.3 Allocating Logical Functions to Hardware Architecture Elements

As already discussed in Subsection 4.2, a model–based development process usually involves creating models for different perspectives of the system. The structure of the modeled system is not necessarily the same in the various perspectives, which requires means to correlate the different models to each other. Here we concentrate on the *logical* and *technical* perspective and the relationships among the contained models. As Figure 4 shows, we refined the initial logical functional specification and its requirements specified as contracts. Note that this is still part of the *logical perspective* of the system. On the right hand side of Figure 4, an excerpt of the model of the technical architecture is depicted along with an exemplary allocation of an instance of the function *Monitor* to a *Task*, that is scheduled with other tasks inside a partition of a hierachical scheduler running on an electronic control unit (ECU). This modeling–example also illustrates the benefit of the concept of perspectives as they allow to separate different design concerns. The functional specification of a *WheelBrakeControlSystem* is, at least in the early stages of development, independent of the underlying plattform hosting and executing the functions.

Note that the CSM also provides means to specify properties of tasks and their schedulers such as priorities, execution times and scheduling policies. But as this is outside the scope of this paper these features are only briefly mentioned here.

5.4 Semantics of Realize and Allocation

So far we have introduced the concepts of abstraction levels and perspectives and illustrated their usage by an example. However, the power of CSM lies in its rigid semantic foundation allowing to apply model–checking techniques. Thus, we need a definition of *realize* and *allocation* according to these semantics. Figure 5 sketches this definition that relies on *HRC state machines*. The idea is to relate the observable behavior of components exposed at its ports.

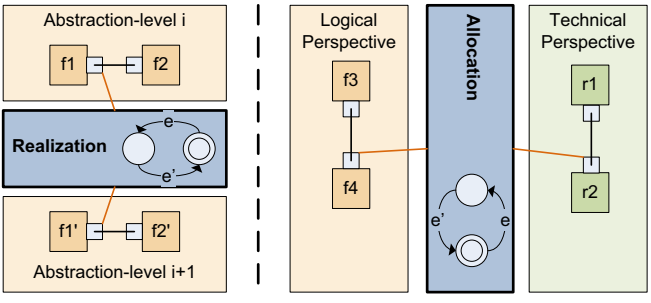


Fig. 5. Semantics of refinement and function allocation

Realization: Realizations are relationships between ports of components on different abstraction levels. Intuitively a realization–link states, that a component (e. g. *f1*) has somehow been refined and is now more concrete in regards to its interface and/or behavior (e. g. *f1'*). This cannot always be captured by a pure decomposition approach. Thus, we define the realization of a component by introducing a state–machine that *observes* the behavior of the refined component *f1'* and translates it into according events, that are observable at a port of component *f1*.

Allocation: Allocations are relationships between ports of components in different perspectives. Intuitively an allocation–link states that the logical behavior of a component (e. g. *f4*) is part of the behavior of a resource (e. g. *r2*), to which it has been allocated. Here we consider the same link–semantics as for the realization: It is defined as a state–machine that *observes* the behavior of the resource *r2* and translates it into according events that are observable at a port of the allocated component *f4*.

6 Conclusion

A new methodology to support the development process of safety–critical systems with contracts has been described in this paper. We first compared existing meta–models also stating their short–comings in relation to our approach. Then we introduced HRC as the semantic foundation of our meta–model. In Section 4 we described our concepts of abstraction levels, perspectives, and viewpoints.

We described the transition concepts between models in Section 5 along with an example.

It is hard to measure (in numbers) how well our model–based methodology competes with other approaches. But we think, from this work it can be seen how valuable these concepts can be to support a development process. Especially the realize and allocate relations deserve more in–depth research in the future since they hold the key to a persistent system design process.

References

1. The ATESST Consortium. EAST ADL 2.0 Specification (February 2008)
2. AUTOSAR. Specification of Timing Extensions, Version 1.0.0 (November 2009)
3. AUTOSAR GbR. Technical Overview, Version 2.2.2 (August 2008)
4. Broy, M., Feilkas, M., Grünbauer, J., Gruler, A., Harhurin, A., Hartmann, J., Penzenstadler, B., Schätz, B., Wild, D.: Umfassendes Architekturmodell für das Engineering eingebetteter oftwareintensiver Systeme. Technical report, Technische Universität München (May 2008)
5. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The Architecture Analysis & Design Language (AADL): An Introduction. Carnegie Mellon University, Pittsburgh (2006)
6. International Organization for Standardization (ISO). ISO 26262: Road vehicles – Functional Safety
7. Josko, B., Ma, Q., Metzner, A.: Designing Embedded Systems using Heterogeneous Rich Components. In: Proceedings of the INCOSE International Symposium 2008 (2008)
8. Meyer, B.: Applying "design by contract". Computer 25(10), 40–51 (1992)
9. Object Management Group. OMG Systems Modeling Language (OMG SysML ™), Version 1.1 (November 2008)
10. Object Management Group. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.0 (November 2009)
11. Project SPEEDS: WP.2.1 Partners. SPEEDS Meta-model Behavioural Semantics — Complement do D.2.1.c. Technical report, The SPEEDS consortium, not publically available yet (2007)
12. Radio Technical Commission for Aeronautics (RTCA). DO-178B: Software Considerations in Airborne Systems and Equipment Certification
13. Society of Automotive Engineers. SAE ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment. Warrendale, USA (December 1996)
14. The CESAR Consortium. CESAR Project, http://www.cesarproject.eu
15. The SPEEDS Consortium. SPEEDS Project, http://www.speeds.eu.com
16. The SPES 2020, Consortium. SPES 2020: Software Plattform Embedded Systems (2020), http://www.spes2020.de
17. Thyssen, J., Ratiu, D., Schwitzer, W., Harhurin, A., Feilkas, M., Thaden, E.: A system for seamless abstraction layers for model-based development of embedded software. In: Proceedings of Envision 2020 Workshop (2010)

Combining Ontology Alignment with Model Driven Engineering Techniques for Home Devices Interoperability

Charbel El Kaed[1,2], Yves Denneulin[2],
François-Gaël Ottogalli[1], and Luis Felipe Melo Mora[1,2]

[1] France Telecom R&D
[2] Grenoble University
charbel.elkaed@orange-ftgroup.com,
yves.denneulin@imag.fr,
francois-gael.ottogalli@orange-ftgroup.com,
luisfelipe.melomora@orange-ftgroup.com

Abstract. Ubiquitous Systems are expected in the near future to have much more impact on our daily tasks thanks to advances in embedded systems, *"Plug-n-Play"* protocols and software architectures. Such protocols target home devices and enables automatic discovery and interaction among them. Consequently, smart applications are shaping the home into a smart one by orchestrating devices in an elegant manner.

Currently, several protocols coexist in smart homes but interactions between devices cannot be put into action unless devices are supporting the same protocol. Furthermore, smart applications must know in advance names of services and devices to interact with them. However, such names are semantically equivalent but syntactically different needing translation mechanisms.

In order to reduce human efforts for achieving interoperability, we introduce an approach combining ontology alignment techniques with those of Model Driven Engineering domain to reach a dynamic service adaptation.

Keywords: SOA, Plug-n-play protocols, ontology alignment, MDE.

1 Introduction

Ubiquitous Systems imagined by Mark Weiser in [21] where computer systems are anywhere and invisible are not that far. Many projects from the industry WRally[1] and academia GatorTech[2] are pushing this vision further. Such systems rely on the service-oriented architecture which provides interactions between loosely coupled units called services.

Discovery, dynamicity and eventing are the main features of service oriented systems that suit best so far ubiquitous systems characteristics. In such systems,

[1] http://www.microsoft.com/whdc/connect/rally/default.mspx
[2] http://www.gatorhometech.com/

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 71–82, 2010.

devices interact with each others and inter-operate transparently in order to accomplish specific tasks.

Smart applications are currently being deployed on Set-Top-Boxes and PC acting as control points by orchestrating home devices such as lights, TV, printers. For example a *Photo-Share* smart application automatically detects an IP digital camera device and, on user command, photos are rendered on the TV and those selected are printed out on the living room printer. The Photo-Share application actually controls such devices and triggers commands upon user request, all the configuration and interaction is completely transparent to the user who only chooses to buy Photo-Share from an application server and deploys it on his home gateway.

Plug-n-Play protocols follow the service-oriented architecture approach where home devices offer services and associated actions, for example a UPnP light device offers a SwitchPower service with two associated actions: SetTarget(Boolean) to turn on or off a light and the GetStatus() action to retrieve the actual state of the light. Currently, UPnP, DPWS, IGRS and Apple Bonjour protocols coexist in smart-homes environments but interactions between devices can not be put into action unless devices are supporting the same protocol. Furthermore, smart applications need to know in advance, names of services and actions offered by devices in order to interact with them. Equivalent device types have almost the same basic services and functions, a printer is always expected to print independently from the underlying protocol it is using. Unfortunately, equivalent devices supporting different protocols share the same semantics between services and actions but not the same syntax for identifying such services and actions. This heterogeneity encloses smart applications into specific and preselected device and service orchestrations.

Smart applications need to be set free from protocol and service syntax heterogeneity. The user must not be restrained to one type of protocol and devices, he should be able to integrate easily and transparently equivalent devices to his home environment. Existing work proposes service interoperability frameworks and techniques which starts by identifying similarities between services and functions. The identification and matching process is performed most of the time manually followed by different techniques to abstract service semantics, the process eventually ends up by applying service adaptability through template based code generation.

In order to reduce human efforts for achieving interoperability we introduce in this article an approach that combines ontology alignment techniques with those of Model Driven Engineering domain to reach a dynamic service adaptation and interaction.

The remainder of the paper is organized as follow: section 2 provides a brief overview of Plug-N-Play protocols. Section 3 overview SOA and OSGi whereas section 4 deals with Ontology Alignment and MDE techniques. Section 5, 6 describes our approach and its implementation. Section 7 discusses relevant related works. Eventually, Section 8 draws conclusions and outlines future works.

2 Plug-N-Play Protocols

UPnP [20], IGRS [10], Apple Bonjour [2] and DPWS [17], the newly standardized protocol supported mainly by Microsoft and included in Windows vista and 7, cohabit in home networks and share a lot of common points. They are all service-oriented with the same generic IP based layers: addressing, discovery, description, control and eventing. They also target the same application domains, multimedia devices are shared between UPnP, IGRS and Bonjour while the printing and home automation domains (printers, lights) are dominated by UPnP and DPWS.

Each protocol defines standard profiles specifying required and optional implementation that manufacturers need to support. Of course, vendors can extend the standards using specific notations and templates.

Even though those protocols have a lot in common, devices cannot cooperate due to two main differences:

- Device Description: expose general device information (id, manufacturer, model etc), supported service interfaces along with associated action signatures and parameters. However, equivalent device types support the same basic functions which are semantically similar but syntactically different. For instance, on a DPWS light [19], Switch(Token ON/OFF) is semantically equivalent to the SetTarget(Boolean) on a UPnP light [20]. This heterogeneity prevents smart applications to use any available device, regardless of their protocol, to accomplish a certain task such as printing or dimming lights.
- The IP based layers: plug-n-play protocols define their own underlying protocols each adapted to its environment. UPnP and IGRS uses GENA[3] for eventing, SSDP[4] for discovery and SOAP for action invocation while DPWS is based on a set of standardized web services protocols (WS-*). DPWS uses WS-Discovery and WS-MetaDataExchange to discover a device, WS-Eventing for notifications and SOAP for action invocations. WS-Security, WS-Policy are used to provide secure channels during sessions. Apple Bonjour uses multicast-DNS. Obviously, this heterogeneity between such protocols increases the complexity of device interoperability starting from the IP based layers.

  Thus, most interoperability frameworks propose a centralized approach using specific protocol proxies rather than a P2P interactions between home devices.

3 Service Oriented Architecture

The service-oriented architecture is a collection of entities called services. A provider is a service offering one or more actions invocable by entities called service clients. Every service is defined by an interface and a signature for each provided action. The service registry is an entity used by service providers to

[3] General Event Notification Architecture.
[4] Simple Service Discovery Protocol.

publish their services and by the clients to request available services in an active (available providers) or passive discovery mode (listens to service registration events). The service interface and other specific properties are used to register and request services in the service registry. The service client must know in advance the interface name of the requested service. Since the request is based on the interface name, a client requesting an interface with a semantically equivalent but syntactically different name will not receive the reference of a similar service.

OSGi, as an example, is a dynamic module system based on the service-oriented architecture. An OSGi implementation has three main elements : Bundles, Services, and the Framework. A bundle is a basic unit containing Java classes and other resources packaged in (.jar) files which represent the deployment units. A Bundle can implement a service client, a provider or an API providing packages to other bundles. The framework defines mechanisms to manage dynamic installation, start, stop, update, removal and resolution of dependencies between bundles. Once a bundle dependency is resolved, it can be started and the service can interact with other services.

In [4], **Base Drivers** are defined as a set of bundles enabling to bridge devices with specific network protocols. A base driver listens to devices in the home network then create and register on the OSGi framework a proxy object reifying the founded device. Services offered by real devices on the home network can now be invoked by local applications on the OSGi framework. The device reification is dynamic, it reflects the actual state of the device on the platform. The local invocation on OSGi is forwarded to the real device.

Currently, there is a UPnP base driver[5] implementation published by Apache, a DPWS and Apple Bonjour Base Drivers previously developed in our team [3], [4] and a DPWS base driver proposed by Schneider [19]. Base drivers solve the IP based layers heterogeneity of Plug-n-Play protocols (see section 2), but the device and service description ones remain.

4 Ontology Alignment and Model Driven Engineering

According to [12], *"An ontology is an explicit representation of a shared understanding of the important concepts in some domain of interest"*. In our work, the domain of interest is the home network and concepts of the ontology are devices, services, actions and parameters. Every real device is modeled by an ontology reflecting its specific information with the predefined concepts.

Alignment [7] is the process of finding a set of correspondences between two or more ontologies, for example finding equivalent services, actions and parameters between a UPnP and a DPWS Light Device. The correspondence between entities is expressed using a normalized similarity value within an $\mathbb{R}^+[0,1]$ interval.

Model Driven Engineering is a software development methodology based on different levels of abstraction aiming to increase automation in program development. The basic idea is to abstract a domain with a high level model then to transform it into a lower level model until the model can be made executable

[5] http://felix.apache.org/site/apache-felix-upnp.html

using rules and transformation languages like in template-based code generation tools. The Model Driven Architecture [18] promoted by the Object Management Group (OMG) is considered as a specific instantiation of the MDE approach. It defines standard models like UML and MOF. MDA defines four architectural layers: M0 as the instance layer, M1 for the model, M2 for the meta-model and the M3 for meta-meta-model layer.

5 Combining MDE and Ontology Alignment Techniques

Our approach is based on two techniques for device adaptability. The first one uses ontology alignment to identify correspondences semi-automatically between equivalent devices. The output of the alignment is then used as a transformation language in order to automatically generate a proxy device which will receive service invocations and adapt them to the equivalent device. All the adaptation process is transparent to the orchestrating application and the existing devices. The proxy will actually bridge syntactic heterogeneity between semantically equivalent device types and services.

Our approach follows four major steps to accomplish device interoperability:

5.1 Ontology Representation

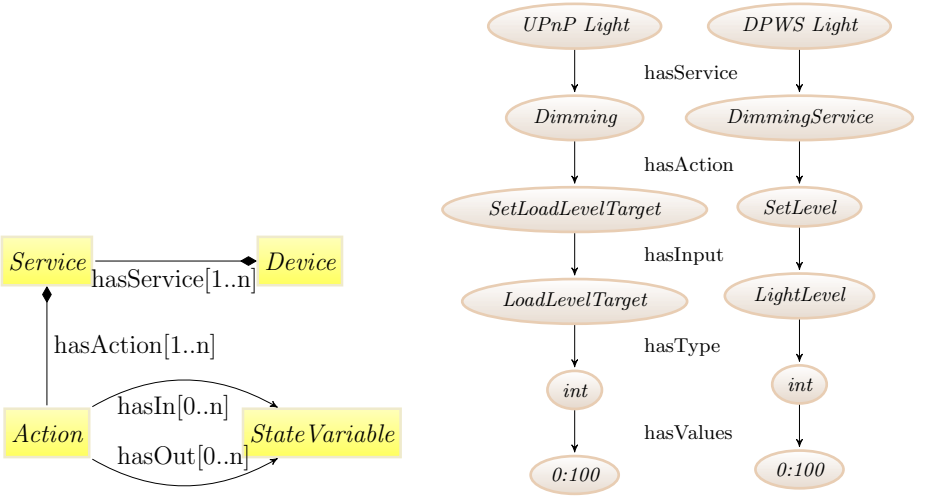
The first step aims to hide device description heterogeneity by modeling each device with common concepts using an ontology. We chose to use common concepts based on the UPnP description model. Every device is modeled with an ontology which is conformed to the meta ontology described in fig.(1 a). We use the concepts as follows: every device has one or more services, every service has one or more actions and each action has one or more input/output state variables. Now we can build device ontologies conformed to this meta ontology.

In fig.(1 b) we present UPnP and a DPWS Device lights ontologies. (For simplicity we omitted from figure (1 b) class types, device, service and properties such as service Id, name, etc).

We automated the construction of device ontologies by using specific OWL Writer bundles fig.(2 a) that listens to the appearance of Plug-N-Play devices, if the device ontology was not yet build for a device type and model (check the device type, model, services etc) then the build process can proceed and once terminated the Aligner is notified.

5.2 Ontology Alignment

The MDA defines 4 levels of architecture, the meta device ontology is on the M2 layer while the instantiated ontologies reflecting home devices are on the M1 layer. Transformation models in the MDE aim to build bridges between models linking entities between two existing models. Those bridges are actually transformation rules written manually or generated using graphical tools using specific languages like ATL [11]. Fig. 3 shows the overview of the approach. We

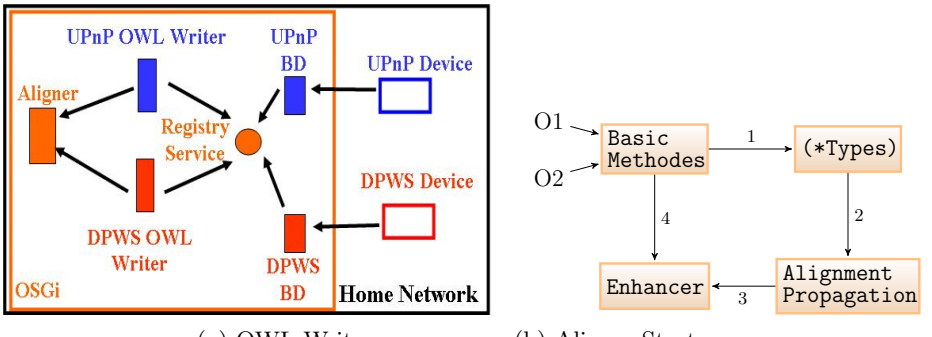


(a) M2: Meta Model Ontology (b) M1: UPnP and DPWS Light ontology

Fig. 1.

use ontology alignment techniques to match entities in a semi-automatic way fig.(2 b) by applying heuristics to match entities of the ontologies.

Step 1 of the aligner takes two ontologies $O1$ and $O2$ then apply basic matching techniques described in ([7], chapter 4) such as Hamming, Leveinshtein, smoa and other techniques based on an external dictionary *WordNet*. We propose and implement an enhanced smoa based technique using wordNet to detect antonyms (Set $\neq$ Get, Up $\neq$ Down etc) and provide more accuracy during action matching like "SetLoadLevel" $\neq$ "GetLoadLevel". Basic techniques provides those two actions as a potential match while smoa++ clearly reduces the similarity value between these two strings. The first step matches all concepts, it uses all the



(a) OWL Writers (b) Aligner Strategy

Fig. 2.

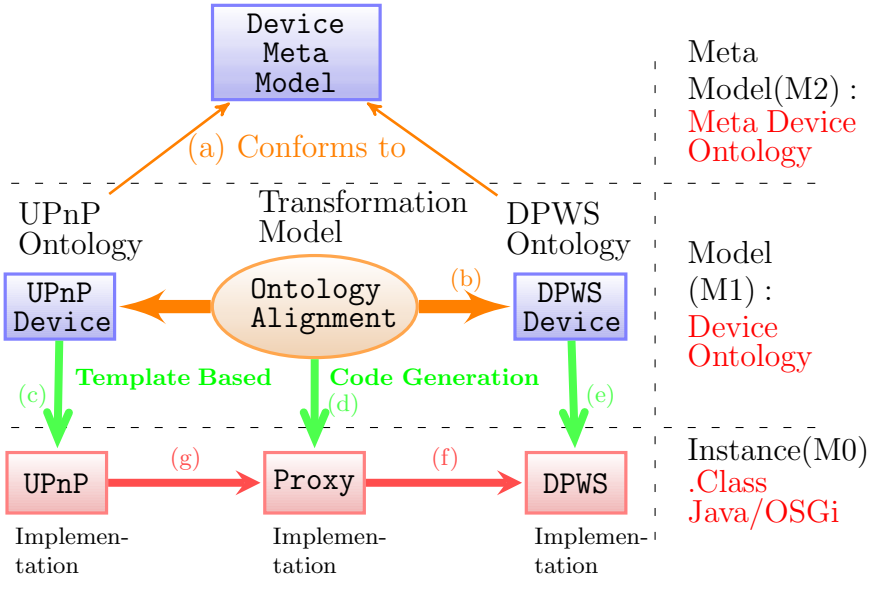


Fig. 3. Overview of the approach

available information in the ontology. We found in some cases that there is a higher similarity between a service name and an action name, we will use this information to enhance similarities in the step 4 of the aligner.

Since each basic technique has its own advantages and disadvantages, we combine all these techniques and use a weight α_k for each method k, $\alpha_k \in \mathbb{R}^+[0,1]$. We give a higher weight for methods using an external dictionary.

Step 2 of the aligner filters the alignment by applying typed classes, it keeps same concept type correspondences (device-device, service-service, etc), we use $\beta_{i,j} = 1$ if concept i and j are the same and zero if not. The output of the first and second steps is a matrix of similarities between concepts.

$Sim1_{i,j,k}$ is the similarity result after step 1 between entity i from $O1$ and entity j from $O2$ using the technique k. $Sim2_{i,j,k}$ is the output after step 2.

$$Sim2_{i,j} = \left[\sum_{k=1}^{n} (\alpha_k * Sim1_{i,j,k})\right] * \beta_{i,j} \in \mathbb{R}^+[0,1] \quad (1)$$

Depending on the devices complexity, we can choose to trim alignments and keep all similarities higher than a *threshold* and additionally use a Delta Threshold δ to keep matched concepts having a similarity differing at most by a tolerance value δ.

Step 3 of the aligner propagates similarities along the ontology structure, for example when two services have strong similarity, we enhance the similarity of their actions. Our method is a *Down Propagation* method, based on Coma++ [6] which uses an *Up Propagation* method.

Step 4 aims to enhance similarity values, if a $Service_A$ have a high similarity with $Action_B$ of $Service_B$, then the algorithm enhances the similarity between both services.

Step 5 is actually the human intervention to validate or edit correspondences between entities. The output, expressed in OWL, represents the transformation rules for the next step to automatically generate the device proxy with the right matching between devices.

In our approach, UPnP is chosen as a common model, UPnP services are substituted with non-UPnP devices. Other services are matched with UPnP services and if a match is found then we can adapt home applications on the fly to invoke non-UPnP existing services. Our choice is motivated by the wide acceptance of UPnP among device manufacturers and telecommunication operators, and the large number of devices standardized by DLNA (www.dlna.org). Those characteristics makes UPnP the most mature protocol so far among existing plug-n-play protocols and therefore the best pivot candidate for our approach.

5.3 Template-Based Code Generation

The input of this step is the ontology alignment between two devices. Since we have the correspondences between elements of the ontology then the proxy can be generated using already written templates. This process is already used to generate Java code from device description UPnP XML description and WSDL files to Java code (interfaces and classes), then it is up to developers to implement the necessary functional code (fig. 3, c and e arrows). In our approach, all the code is actually automatically generated since the correspondences between devices are provided by the ontology alignment output, fig.(11,b). The templates need only to be filled with the aligned device, service and action names along with parameters. When the application invokes the UPnP dimming action *SetLoadLevelTarget* with the parameter *LoadLevelTarget*, the proxy will invoke on the DPWS device the *SetLevel* action of the *DimmingService* service and *LightLevel* parameter as an input.

We identified another case where an action is an union of two others, such is the case of the standard UPnP and DPWS printers [20,13]. The UPnP action *CreateJobV2* (simple entries) is equivalent to two DPWS Actions *CreatePrintJob* (complex structured entries) and *AddDocument*. The mapping between DPWS. CreatePrintJob and UPnP.CreateURIJob reveals that the parameter *SourceURI* has an equivalent entry parameter *DocumentURL* for the DPWS action *AddDocument*. Consequently, CreateURIJob = UnionOf (CreatePrintJob, AddDocument). Other properties exist too, such as Sequenced-UnionOf where actions have input and output dependencies. This kind of properties can be detected easily using a reasoner to infer on correspondences between services and actions. Other actions on the printer devices are substitutable such as (UPnP.GetJob Attributes & DPWS.GetJobElements), (UPnP.GetPrinterAttributesV2 & DPWS. GetPrinterElements) and (UPnP.CancelJob & DPWS.CancelJob). These equivalent actions make the standard UPnP and DPWS printer devices substitutable.

Table 1. Mapping between a standard DPWS and a UPnP printer action

DPWS (CreatePrintJob)	UPnP (CreateURIJob)
PrintTicket/JobDescription/JobName	JobName
PrintTicket/JobDescription/JobOriginatingUserName	JobOriginatingUserName
PrintTicket/JobProcessing/Copies	Copies
X	SourceURI
PrintTicket/DocumentProcessing/NumberUp/Sides	Sides
PrintTicket/DocumentProcessing/NumberUp/Orientation	OrientationRequested
PrintTicket/DocumentProcessing/MediaSizeName	MediaSize
PrintTicket/DocumentProcessing/MediaType	MediaType
PrintTicket/DocumentProcessing/NumberUp/PrintQuality	PrintQuality

5.4 Service Adaptation

Now that all elements are ready, the proxy can be generated on demand and compiled on the fly in order to provide interoperability between devices, services and actions. For the *à la carte* application, the user can choose the DPWS (or other) device type and model, then based on the existing specific code templates and the ontology alignment on the service provider platform, the proxy can be generated and packaged for deployment. As for the on the fly adaptability, the service request will be intercepted. The correspondent proxy will be generated according to ontology alignment already existing on the gateway or downloaded from the service provider servers. Thus the Photo-Share application will work transparently for the user.

6 Implementation

In this work in progress, we used UPnP felix Apache[6] and DPWS [19] base drivers. We developed OWL Writers on an Felix/OSGi framework using the OWL API and implemented our alignment strategy using the Alignment API [8]. The output of the alignment is expressed with *OWLAxiomsRendererVisitor* which generates an ontology merging both ontologies to express relations between entities. To subsume properties like Union-Of we will use a reasoner on the output to infer such properties. The proxy is currently generated using manually pre-filled templates using Janino[7] on the fly compiler. In future work, we will fill the templates with the information from the ontology alignment.

7 Related Work

Different approaches have been developed, in the literature, to solve the issues related to the interoperation problem, the approaches can be put in three major

[6] http://felix.apache.org/site/apache-felix-upnp.html
[7] www.janino.net

categories : EASY [1] and MySIM [9] worked on frameworks allowing services to be substituted by other similar services and actions. They both model the domain in a **common ontology** holding all the concepts and properties relating them. Every service interface, action and parameter is annotated with a predefined semantic concept from the common ontology. A service is then substitutable by another if actions and their input/output parameters have similar or related semantic concepts from the common ontology. MySIM uses Java introspection in order to retrieve the annotations then compare semantic concepts and adapt the matched services and actions. The limitation of both approaches is that annotations are filled with predefined concepts, consequently when a new service type appears, the common ontology should be updated first by adding new concepts and connecting them to other existing entities in the ontology. The update process is not that obvious since a new type can have common semantics with more than one existing concept resulting an incoherent ontology. Consequently, a reorganization operation is more often required to reestablish a coherent classification between concepts in the common ontology.

The second category models the domain with an **abstract representation** like an ontology in [5] or with a meta model in [16]. In DOG, similar device types and actions are modeled with abstract ontology concepts, (light device, dimming action, switch on/off) then these concepts are mapped to specific technology actions and syntax using specific rules to fill pre-written Java/OSGi code templates. The interoperability among devices is based on abstract notifications messages and predefined association between commands (the switch OffNotification is associated to the OffCommand on a device light). Then MVEL rules are generated automatically based on the manually written associations, the rules actually invokes technology specific functions on the targeted device. EnTiMid [16] uses a meta model approach instead of the ontology then generates specific UPnP or DPWS Java/OSGi code using transformation rules and templates. Both approaches deals with relatively simple devices like lights with similar actions and parameters. However the abstraction of complex devices like printers is not trivial specially when an action on one device is equivalent to one or more actions on another equivalent device (Table 1). Besides, in both approaches, the mapping between the abstract model and the specific model is done manually by either writing transformation rules or writing predefined associations and automatically generating transformation rules.

The third category uses a **common language** to describe devices with same semantics, Moon et al. works on the Universal Middleware Bridge [15] which proposes a Unique Device Template (UDT) for describing devices. They maintain a table containing correspondences between the UDT and the Local Device Template (LDT). UMB adopts a centralized architecture similar to our approach, where each device/network is wrapped by a proper UMB Adapter which converts the LDT into UDT. Miori et al. [14] defines the DomoNet framework for domotic interoperability based on Web Services and XML. They propose DomoML a standard language to describe devices. TechManagers, one per subnetwork translates device capabilities as standard Web Services, the mapping is

done manually. Each real device is mirrored as a virtual device in other subnetworks, thus each device state change requires considerable synchronization effort among subnetworks. The HomeSOA [4] approach uses specific base drivers to reify devices locally as services then another layer of *Refined drivers* abstract service interfaces per device types as a unified smart device, a UPnP and DPWS light dimming services are abstracted with DimmingSwitch interface then it is up to the developer to test the device type and invoke the underlying specific interface.

8 Conclusion and Future Works

In this article we propose an approach based on MDE and ontology alignment techniques to bridge device and service syntax heterogeneity in order to enable service interoperation. First, we automatically generate for each device an ontology conformed to a meta ontology, then we apply ontology alignment techniques to semi-automatically retrieve correspondences between equivalent device types and services. The ontology alignment is validated by a human. The resulting output corresponds to transformation rules used in oder to generate on the fly specific proxies to enable interoperability. We choose UPnP as a central and pivot protocol. We match other protocol services with those from UPnP. This choice is motivated by the fact that UPnP is the most mature protocol among plug-n-play protocols so far. There is a large number of standardized devices (www.dlna.org) and it is widely accepted and supported by many manufacturers and vendors. The proxy is generated when there is equivalent device type to a UPnP requested device. The specific proxy publishes UPnP service interfaces and actions and once invoked it actually transfer the invoked action to the correspondent device using its own semantics and syntax. In future works, a GUI will be implemented to make the validation and correspondence editing easier. Complex mapping of services and actions (Sequential and simple Union-Of relations) will be investigated in order to infer and deduce such correspondences with a minimal human intervention.

Acknowledgment

The authors would like to thank Sylvain Marie from Schneider Electric for his help with the DPWS Base Driver integration.

References

1. Ben Mokhtar, S., et al.: Easy: Efficient semantic service discovery in pervasive computing environments with qos and context support. Journal of Systems and Software (2008)
2. Bonjour, http://www.apple.com/support/bonjour/
3. Bottaro, A.: Rfp 86 - dpws discovery base driver (2007), http://pagesperso-orange.fr/andre../rfp-86-DPWSDiscoveryBaseDriver.pdf

4. Bottaro, A., Gérodolle, A.: Home soa -: facing protocol heterogeneity in pervasive applications. In: ICPS '08: Proceedings of the 5th International Conference on Pervasive Services (2008)
5. DOG: Dog: Domestic osgi gateway, http://elite.polito.it/dog-tools-72
6. Engmann, D., Maßmann, S.: Instance matching with coma++. In: BTW Workshops (2007)
7. Euzenat, J., Shvaiko, P.: Ontology Matching. Springer, Heidelberg (2007)
8. Euzenat, J.: Alignment api, http://alignapi.gforge.inria.fr
9. Ibrahim, N., Le Mouël, F., Frénot, S.: Mysim: a spontaneous service integration middleware for pervasive environments. In: ICPS '09 (2009)
10. IGRS: http://www.igrs.org/
11. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A model transformation tool. Science of Computer Programming 72, 31–39 (2008)
12. Kalfoglou, Y.: Exploring ontologies. In: Chang, S.K. (ed.) Handbook of Software Engineering and Knowledge Engineering. Fundamentals, vol. 1 (2001)
13. Microsoft: Standard dpws printer and scanner specifications (January 2007), http://www.microsoft.com/whdc/connect/rally/wsdspecs.mspx
14. Miori, V., Tarrini, L., Manca, M., Tolomei, G.: An open standard solution for domotic interoperability. IEEE Transactions on Consumer Electronics (2006)
15. Moon, K.d., et al.: Design of a universal middleware bridge for device interoperability in heterogeneous home network middleware. IEEE Transactions on Consumer Electronics (2005)
16. Nain, G., et al.: Using mde to build a schizophrenic middleware for home/building automation. In: ServiceWave '08: Proceedings of the 1st European Conference on Towards a Service-Based Internet (2008)
17. OASIS: Devices profile for web services version 1.1 (2009), http://docs.oasis-open.org/ws-dd/dpws/1.1/os/wsdd-dpws-1.1-spec-os.html
18. Object-Management-Group, O.: Mda guide version 1.0.1 (2003)
19. SOA4D: Service oriented architecture for devices, https://forge.soa4d.org/
20. UPnP: http://www.upnp.org/
21. Weiser, M.: The computer for the 21st century. SIGMOBILE Mob. Comput. Commun. Rev. 3(3), 3–11 (1999)

Rewriting Logic Approach to Modeling and Analysis of Client Behavior in Open Systems

Shin Nakajima[1], Masaki Ishiguro[2], and Kazuyuki Tanaka[3]

[1] National Institute of Informatics, Tokyo, Japan
[2] Mitsubishi Research Institute, Inc., Tokyo, Japan
[3] Hitachi Software Engineering Co., Ltd., Tokyo, Japan

Abstract. Requirements of open systems involve constraints on clients behavior as well as system functionalities. Clients are supposed to follow policy rules derived from such constraints. Otherwise, the system as a whole might fall into undesired situations. This paper proposes a framework for system description in which client behavior and policy rules are explicitly separated. The description is encoded in Maude so that advanced analysis techniques such as LTL model-checking are applied to reason about the system properties.

Keywords: Requirements, Validation, Model-Checking, Maude.

1 Introduction

Embedded and ubiquitous computing systems are typically open, and their functional behavior has much impact from the environment [5]. It involves human behavior if the systems are user-centric. Unfortunately, thoroughly anticipating client behavior is a hard task and a certain assumption on the client behavior is usually made. Clients are expected to follow an operation guideline to consist of policy rules, which is a concrete form of such assumptions.

In general, a relationship between requirements and system specification is concisely summarized in the *adequacy* property [4]; $D \wedge S \Rightarrow R$, where R and S represent requirements and system specification respectively. D is *Domain Knowledge* that provides background information and is an assumption on the system. D, in open systems involving clients, can be elaborated into $D_c \wedge D_p \wedge D_o$. D_c and D_p refers to the domain knowledge on client behavior and policy rules respectively while D_o represents other domain knolwedge and is omitted here for brevity.

Requirements R may usually consist of typical usecases to assume that clients behave in a *regular* manner ($D_c \wedge S \Rightarrow R$). Clients, however, sometimes show irregular behavior. Some may be careless, and a few might even try to cheat the system or obstruct other clients. Such behavior could result in undesired consequence. To avoid such situations, the policy rules (D_p) are introduced to constrain the client behavior. There is, however, another problem; it cannot be ensured whether clients follow the policy rules or not. In a word, the assumption made at the time of the system development is no longer valid at the time of

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 83–94, 2010.

operation. It is desirable to provide means fo studying possible outcomes at the development time.

In this paper, we will discuss an approach to requirements modeling of open systems involving human clients. The proposed method allows us to represent and reason about the system specification at early stages of the development. It makes it possible such an analysis as whether a certain irregular client behavior might lead to situations against the requirements. In particular, the contribution of this paper is summarized as follows.

1. it proposes a framework for system description in which clients behavior (D_c) and policy rules (D_p) are explicitly separated,
2. the description is encoded in Maude [3], thus allowing automated analysis.

The rest of the paper is structured as follows. Section 2 introduces Guiding System as a motivating and running example. Section 3 presents a formalization approach to the problem and a role of model-checking in the analysis. Section 4 explains the encodings in Maude. Section 5 shows a case study, which is followed by discussions and conclusion in Section 6.

2 Guiding System: A Motivating Example

A Guiding System is used so as to illustrate the problem and approach. In Figure 1, the Guiding System is a user-navigation system for clients to access particular resources. Clients arriving at the system firstly check with their ID Card if they have been registered. They then wait in a Waiting Room until receiving a call to allow them to exclusively use the resource. The system keeps track of the status information of the resource usage and waiting queue of clients. When a resource is free, the system will call a client in the Waiting Room. The client moves to the Work Room where he can access the resource.

Except the above regular scenario, a lot of cases could happen. A client might try to go to the Work Room even if he was not the one to be called. A client might walk around in the Waiting Room disturbing others by blocking the door

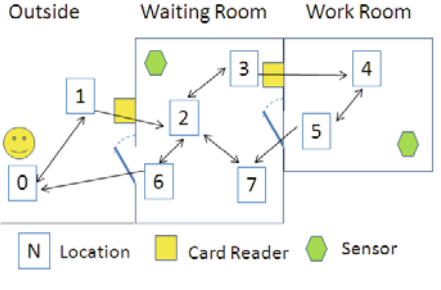


Fig. 1. Guiding System

to the Work Room. Some might be absent when he is called, and would not use the resource at all.

The Guiding System might detect some of the anomalous situations if certain fancy gadgets are available. For example, all the client locations can be monitored if every person has a remote badge that can be sensed with wireless communication network. Precise location of each client is surely monitored. It, however, will increase the cost of the system.

Alternatively, the Guiding System is used with a certain running policy, which poses some constraints on the client behavior. In the real world, an officer works near the system and he periodically checks to make sure all the clients behave in a regular manner. For example, he will tell a client who walks around in the Waiting Room, "please sit down and wait for a call." The system can be defined without being over-engineered since the clients are assumed to follow mostly the regular scenario. The implicit assumptions on the client behavior are made explicit in the form of policy rules.

3 Formal Analysis of Client Behavior

3.1 Basic Computation Model

As discussed in Section 2, what will be specified consists of many entities executing concurrently. Some entities constitute the Guiding System, and clients are autonomous to move in the rooms. Furthermore, a hypothetical component to enforce the policy rules executes independently from other entities. Such observations lead us to adapting a modeling method of using a set of communicating state-transition machines. Additionally, we have introduced a notion of Location Graph to reason about the client movement. Below are some definitions to constitute the basic model.

[**Extended Mealy Machine**] Extended Mealy Machine (EMM) M is intuitively an object to have its own internal attributes and its behavioral aspect specified by a finite-state transition system. They communicate with each other by sending and receiving events. An EMM M is defined as a 6-tuple

$$M = (Q, \Sigma, \ \rho, \delta, q_0, \mathcal{F})$$

Q is a finite set of states, Σ is a finite set of events, q_0 is an initial state, and $\mathcal{F}$ is a finite set of final states. Variable map ρ takes a form of $Q \rightarrow 2^V$ (a set of attribute values). Transition relation δ takes a form of $Q \times A \times Q$. The action A has further structure to have the following functions defined on it; $in : A \rightarrow \Sigma$, $guard : A \rightarrow L$, $out : A \rightarrow 2^\Sigma$ where L is a set of predicates.

Operationally, a transition fires at a source state q to cause a state change to a destination p if and only if there is an incoming event $in(A_q)$ and $guard(A_q)$ is true. In accordance with the transition, possibly empty set of events $out(A_q)$ are generated. Furthermore, *update* function is defined on A to change ρ, which modifies the attribute values stored in M.

[**Configuration**] Configuration constitutes a set of EMMs and a communication buffer to store events (Σ) used for asynchronous communication between EMMs.

[**Run**] Run of a EMM$^{(j)}$ ($\sigma^{(j)}$) is a sequence of states ($Q^{(j)}$) which are obtained by a successive transitions from the initial state ($q_0^{(j)}$) following the transition relation $\delta^{(j)}$. A run of Configuration is a possible interleaving of $\sigma^{(j)}$ for all EMMs, taking into account of the event communications among them.

3.2 Location and Client Behavior

[**Location Graph**] Location Graph is a directed graph (N, E). N is a set of locations, and a location is where clients can occupy; clients walk from a location to another. Since clients walk along the edges between locations, nodes N are connected with edges E, which together form a directed graph.

As seen in Figure 1, clients use doors to enter and leave a room. Such doors are considered to attach with particular edges where clients can move along. Location Graph, in this sense, represents the structure of the rooms that the system works on.

Note that location in the graph does not correspond to a physical place in the real world, but represents an abstract place corresponding to a set of them. It is abstract in that client behavior from any of the pysical places in the set is not distinguished.

[**User Machine**] User Machine (UM) is a special EMM, which has a default attribute to represent its location (N of Location Graph). Furthermore, client behavior is designed to follow a simple template. First, it is in *Mediate* state to determine which location it moves to next (*a trial step*). Second, a state transition is fired to *Walk*, in which the client actually changes its location. Third, another transition puts the client to *Action* state to take various actions such as swiping his ID Card or opening doors.

In each state, UM takes a sequence of *micro-transitions* between *micro-states* to show application-specific behavior. Therefore, a run for User Machine consists of micro-states, but a sequence of Locations is what we want to know.

[**Location Projection**] Location Projection ξ is a function to extract a pair of client Id and its Location from state Q of User Machine EMM; $\xi : Q \rightarrow \text{ClientId} \times N$. Note that ξ can work only on User Machine. It returns ϵ (*null*) for all the other EMMs.

[**Location Trail**] Location Trail is a sequence of location projection, which is obtained from a (configuration) run to be a sequence of $\xi(\sigma^{(j)})$.

3.3 Policy Rules

We focus on reasoning about client movements, namely how each client moves around. Policy rule is defined on Location Trail.

[**Policy Rule**] Policy Rule monitors clients movement and enforces them to move as anticipated. It keeps track of Location Trail, and it checks whether a client trial step is admissible or not. A Policy Rule takes the following form.

if $filter(x,\ l)$ **then** $action(x)$

where x and l refer to a client and a location trail respectively. Currently, three forms of actions are considered; (a) **a permission rule** to allow the client as he wants to do, (b) **a stop rule** to instruct the client not to take any action, and (c) **a coercion rule** to make the client to move as the policy rule instructs.

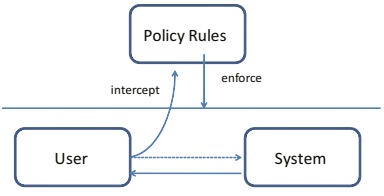


Fig. 2. Two-tiered Framework

[**Policy Enforcer**] Policy Enforcer is an executing entity who enforces a given set of Policy Rules. Policy Enforcer is represented as a special EMM, but executes at a kind of *meta level*. Since it uses Location Trail and checks every trial step, a new architectural mechanism is needed to monitor all the clients movement trial. In Two-tiered Framework shown in Figure 2, user trial step is conceptually intercepted and checked againt Policy Rule, and then its result is enforced. The client moves as the policy rule specifies.

3.4 Analysis Problem

Policy Enforcer, introduced in the previous section, is defined as an entity to monitor the client behavior at runtime to see whether some undesired situation might happen. It basically checks if Location Trail for a particular client falls into anomalous loop, which shows that the client moves indefinitely between some particular locations and that he never goes to a desired destination. When a client rests at a particular location never to move, the anomalous situation is detected as a loop to contain a single location only.

Fig. 3. Lasso Shape

Such anomalous situation is detected as an existence of Lasso Shaped Location Trail (Figure 3). The analysis problem of detecting such a loop can be considered as a model-checking problem [2]. Let $at(C, L)$ be a predicate to represent that a client C is located at a location L. A model-checking problem with given client $u1$ and location $L4$ is written below.

$Configuration \models$ []<>$(at(u1, L4))$

The property to check is read "it is always the case that eventually $at(u1, L4)$ becomes *true*." If the client $u1$ behaves in a irregular manner to form a loop not to reach the location $L4$, a counterexample trace takes a form of the lasso shape as expected.

In other word, the adequacy property [1], $D_c \wedge D_p \wedge S \Rightarrow R$, is here considered as a model-checking problem of $D_c \wedge S \models r$ where r is an LTL (Linear Temporal Logic) formula chosen from R. If r is satisfied, then D_p is not needed. Otherwise, a counterexample trace is obtained, from which policy rules (D_p) are derived. Namely, with D_p so constructed, the relationship $D_c \wedge D_p \wedge S \models r$ holds.

4 Rewriting Logic Approach

4.1 Rewriting Logic and Maude

Rewriting logic is proposed by Jose Meseguer, which follows the tradition of algebraic specification languages [3]. The logic extends order-sorted algebra of OBJ3 to provide means to describe state changes. Maude, an algebraic specification language based on Rewriting logic, is powerful enough to describe concurrent, non-deterministic systems. Such systems can be symbolically represented with appropriate level of abstraction. Furthermore, Maude provides advanced means to analyzing properties of system with various state-space search methods such as bounded reachability and LTL model-checking.

4.2 Encoding Extended Mealy Machine in Maude

With Maude, the artifacts that we are interested in are modeled as a collection of entities executing concurrently. Their functional behavior is expressed in rewriting rule, which takes a form of "$lhs \rightarrow rhs$ **if** *cond*." Transition relation δ from q to p in EMM (Section 3.1) is described by rewriting rule;

$$in(A_q) \ \langle\ \rho_q\ \rangle \quad \longrightarrow \quad \langle\ \rho_p\ \rangle \ \ out(A_q) \quad \textbf{if} \ \ guard(A_q)$$

where $\rho_p = update(\rho_q)$.

Encoding EMM in Maude requires further modules to define. Following example is used to show how an EMM is described. Firstly, MACHINE module is a functional module (`fmod`) that provides a basic syntax (term) of EMM.

```
fmod MACHINE is
 sort Machine . sort StateId .  sort TypeId .
 protecting QID . protecting MID .  protecting ATTRIBUTES .
 subsort Qid < StateId .  subsort Qid < TypeId .
 op <_:_|_;_> : MachineId TypeId StateId Attributes -> Machine [ ctor ] .
endfm
```

[1] D_o is ignored for simplicity.

The module MACHINE defines the sort Machine and imports the sorts of Qid (a built-in primitive sort for representing unique Id) and Attributes. It has a constructor which includes unique Id for the instance, the type of this instance, the state and the attributes. Here is a simple example of Machine, a DOOR.

```
extending SOUP .
op makeDoor : MachineId -> Machine [ctor] .
var D : MachineId .
eq makeDoor(D) = < D : 'Door1 | 'lock ; null > .
```

The constructor `makeDoor` is meant to have an initialized instance of Door Machine. Their dynamic behavior, namely state changes, is described in terms of a set of rewriting rules. The state-transition diagram in Figure 4 shows its simple behavior, which is encoded by the rewriting rules below. Each rewriting rule corresponds to a transition in the diagram.

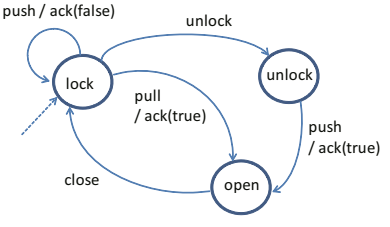


Fig. 4. State Transition Diagram of Door

```
vars D U : MachineId . var R : Attributes .
 rl [1] : push(D, U) < D : 'Door | 'lock ; R >
    => < D : 'Door | 'lock ; R >  ack(U, false) .
 rl [2] : unlock(D) < D : 'Door | 'lock ; R >
    => < D : 'Door | 'unlock ; R > .
 rl [3] : push(D, U) < D : 'Door | 'unlock ; R >
    => < D : 'Door | 'open ; R > ack(U, true) .
 rl [4] : pull(D, U) < D : 'Door | 'lock ; R >
    => < D : 'Door | 'open ; R > ack(U, true) .
 rl [5] : close(D) < D : 'Door | 'open ; R >
    => < D : 'Door | 'lock ; R > .
```

A client generates a `push` event to open a door. If the door is in `lock` state, the rule `[1]` is fired. The door remains shut and sends an `ack(fail)` event to the client. Rule `[2]` shows that the door can be unlocked by the card reader. Alternatively in rule `[3]`, when the door is in `unlock` state, its state is changed from `unlock` to `open` as the door receives a `push` event. The other rules can be understood in a similar manner.

5 Case Study

5.1 Modeling

Location Graph. Figure 5 is an instance of location graph to represent an abstract view of the example in Figure 1. Each location has a capacity attribute to show how many clients are allowed at a time. For example, L1 is where a client swipes his ID Card and its capacity is 1. On the other hand, more than one clients are waiting at L2, thus the capacity of L2 is *many*.

Clients start from L0 and move along the edges depicted with the arrows. At each location, clients may take some actions and they proceed to the next location if prescribed conditions are satisfied.

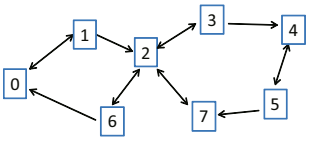


Fig. 5. Location Graph Example

Guiding System. The Guiding System consists of various entities constituting the room (Figure 1), each being defined as an EMM. Figure 6 illustrates all the components and interaction between them. Guide System or GSystem is a main component to provide user-navigation functions for clients to access the resources in the Work Room. It accepts *abstract* events from FrontEnd and makes announcement to let a client enter the Waiting Room.

Door1 is a door to enter the Waiting Room, while Door2 is the one to enter the Work Room. Both is usually locked so that clients must unlock it. The door, however, can be open from the inside to leave even when the door is locked.

CardReader1, when a client swipes his ID Card, checks a backend database to see whether he is a valid registered client. When he is found registered, CardReader1 generates a $unlock(Door1)$ event to unlock the door. CardReader2 also checks the client ID Card when he is called to enter the Work Room. CardReader2 does not access the Database, but uses the valid client information *set* by GSystem. In this way, only the called client is certified.

Figure 7 illustrates the state transition diagram to show behavioral specification of GSystem. Its behavior is defined as a set of responses to three abstract events; *arrive*, *enter*, and *exit*. Their informal meanings are

- $arrive(U)$: a client U enters the Waiting Room from the outside,
- $enter(U)$: a client U enters the Work Room from the Waiting Room,
- $exit(U)$: a client U leaves the Work Room.

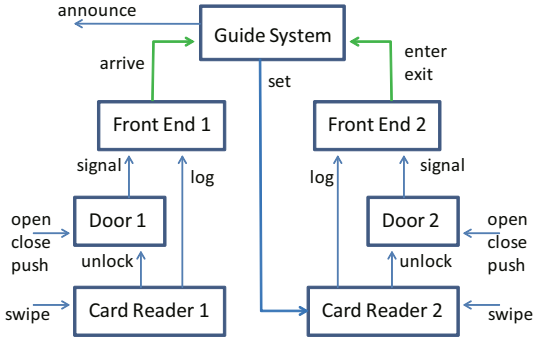


Fig. 6. Guiding System Components and Their Interactions

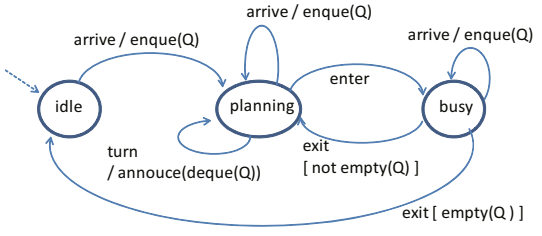


Fig. 7. State Transition Diagram of GSystem

Initially, GSystem is in `idle` state and changes its state to `planning` when an $arrive(U)$ event is generated to indicate that a paricular client U arrives. In the course of the transition, U is enqueued and thus is later consulted for GSystem to determine the next client to enter the Work Room. In view of client, he is put in the queue when he arrives and waits for a call by GSystem.

After generating $announce(U)$ event, GSystem changes its state to `busy` as it receives an $enter(U)$ event. The event indicates that the called client enters the Work Room. All the $arrive(U)$ events are queued while GSystem is in either `planning` or `busy` state. Upon receiving an $exit(U)$ event, GSystem knows that the Work Room is no longer occupied, and it moves to `planning` state again if the queue is not empty. Alternatively, it jumps into `idle` state if the queue is empty.

Although the behavior of GSystem can be defined in terms of three abstract events mentioned above, the instruments equipped with the room do not generate those. Instead, they generate primitive events. For example, CardReader generates $log(U)$ events when U is recognized as a registered client. Door generates *open* and *close* events. There should be a certain mechanism to bridge the gap between the primitive and abstract events.

FrontEnd shown in Figure 6 is responsible for the translation. In particular, FrondEnd1 takes care of the instruments at the entrance of the Waiting Room. It receives $log(U)$, *open*, and *close* events in a specified sequence, and generates

arrive(U) event to GSystem. FrontEnd2, looking at events generated at the border of the Waiting Room and the Work Room. It generates *enter*(U) event from *log*(U), *open*, and *close*. Furthermore, it generates *exit*(U) event from *push* (for leaving the Work Room) and *close*.

5.2 Analysis

Tha Maude description developed so far is composed of 32 modules in about 1,000 lines of codes. About 60 % of codes are responsible for dynamic behavior, namely rewriting rules. Below present some results of analysis conducted with MOMENT Maude 2.4 [1] running on Panasonic CF-W7 under Windows/XP.

Validation in Maude. Firstly, an initial state of the Guiding System shown in Figure 1 is constructed. Secondly, some appropriate User Machine(s) are added to form an initial state. Here, we define a User Machine so that it can initiate its execution by receiving an event *start(U)*. Furthermore, a client follows a regular behavior to leave L0 and return to the same place after doing his job in the Work Room. Below, execution results are represented as Location Trail, traces of a tuple consisting of User and Location, (U, L).

The first example shows a trace of two clients, u1 and u2. The initial state includes two initial events *start(u1)* and *start(u2)*. Here is one possible trace taken from a result of rewriting run (`reduction`) in Maude.

```
('u1, 'L1), ('u1, 'L2), ('u2, 'L1), ('u1, 'L3), ('u2, 'L2), ('u1, 'L4),
('u1, 'L5), ('u1, 'L7), ('u1, 'L6), ('u2, 'L3), ('u1, 'L0), ('u2, 'L4),
('u2, 'L5), ('u2, 'L7), ('u2, 'L6), ('u2, 'L0)
```

As the trace shows, u2 moves to L1 after u1 goes to L2 since L1 is a place to use Card Reader1 and only one user can occupy the location at a time. It also shows that u2 stays at L2 until u1 moves to L6.

The second example consists of a regular client u2 and another client v1. V1, initially at L2, disturbs other users by moving between L2 and L3 repeatedly. Since L3 is a place to allow only one user, no other user can move to L3 while v1 occupies the place. Here is a trace.

```
('u2, 'L1), ('v1, 'L3), ('v1, 'L2), ('v1, 'L3), ('u2, 'L2), ('v1, 'L2),
('v1, 'L3), ('v1, 'L2), ('u2, 'L3), ('u2, 'L4), ('v1, 'L3), ('v1, 'L2),
('u2, 'L5), ('v1, 'L3), ('u2, 'L7), ('v1, 'L2), ('v1, 'L3), ('u2, 'L6),
('v1, 'L2), ('u2, 'L0)
```

Regardless of v1 behavior, the client u2 can move to L2 because more than one clients can share the location. U2, however, has to wait at L2 before going to L3 even he is called. The client v1 moves between L2 and L3, and L3 becomes available occasionally.

LTL Model-Checking. In Section 3.4, we discussed that the monitoring client behavior in view of the Policy Enforcer was a model-checking problem. Model-checking, of course, can be used as *debugging* system behavior as well. Actually, we could find one fault in the original description of the Guiding System where two clients were involved in the scenario. The fault was not revealed just by the validation with `reduction`.

A conflict is occurred at the Door1 (Figure 1). A client u1 at L6 tries to get out of the Waiting Room to push the Door1. Another client u2 swipes his ID card at L1, which is followed by an `unlock` event sent to the same door. It results in a conflict of two events, `push` and `unlock`.

In order to avoid such conflicts, we introduced another Door3, which was used solely as an exit from the Waiting Room. Door1 is now for the entry only. Althought installing a new Door might be a solution to increase the cost of the system, we use the description with Door3 in the following analysis for simplicity. It is especially important to study such design alternatives at this abstract level with the formal analysis of Maude description.

We now demonstrate how model-checking is conducted for the problem relating to Policy Enforcer. Imagine that a client u1 shows irregular behavior. A regular scenario assumes that a client, when he is called at L2, walks to L3 and swipes the ID Card to unlock the door. The client u1, however, walks to L3, but returns to L2 without taking any action at L3, which is repeated indefinitely. Let $goAound(u1)$ be a proposition to denote that the client u1 returns to L0, the outside of the system. The property `[]<>`$(goAound(u1))$ is not satisfied and an obtained evidence is a counterexample, `[L0, L1, L2, L3, L2, L3, ...]`. Such an infinite sequence is abbreviated as `[L0, L1, (L2, L3)*]`, where (...)* represents a loop in the lasso. Policy Enforcer may apply the stop rule (Section 3.3) to the client u1.

Next, consider a case where a non-determinisitc behavior at L3 is added to the above client u1. When he is at L3, he chooses non-deterministically either to walk back to L2 or to enter the Work Room by following prescribed scenario. Model-checking of the property `[]<>`$(goAound(u1))$ returns the same lasso-shaped counterexample as above. We can, however, ensure that there is at least one witness trace for the client to enter the Work Room. The property $\neg$`<>`$(at(u1, L4))$ is not satisfied (namely `<>`$(at(u1, L4))$ is satisfied), and the obtained evidence is a witness of `[L0, L1, L2, L3, L4]`. From the model-checking result, we can derive such a policy rule (D_p) as "clients, once called in the waiting room, should eventually enter the work room." It is an instance of the coersion rule (Section 3.3).

These two example scenarios can be explained in the following ways. Firstly, for a client who shows an irregular behavior (D_c^i), we have $D_c^i \wedge S \not\models$ `<>`$(at(u1, L4))$. Secondly, if the client adds non-determistic behavior (D_c^n) with an appropriate policy rule (D_p) enforced, we have $D_c^n \wedge D_p \wedge S \models$ `<>`$(at(u1, L4))$.

In summary, from the results of $D_c \wedge S \not\models r$, we construct D_p such that $D_c \wedge D_p \wedge S \models r$. Because the aim of D_p breaks the loop in the lasso, a possible method is to adapt the notion of ranking functions [6]. In the most simplifed

form, a ranking function rk is monotonically decreasing with a lower bound lb. D_p is defined to break the lasso loop when rk reaches lb. Policy Enforcer is defined so that it adapts a counter as rk and applies a coersion rule to break the loop when the counter reaches the specified value. Currently, D_p is obtained by a manual inspection of the generated trace.

6 Conclusion

We have proposed a rewriting logic approach to modeling and analysis of client behavior. Firstly, we have discussed the importance of separating human client behavior from policy rules. With a conceptual two-tiered framework, the client can be checked whether his behavior faithfylly follows the policy. Second, such analysis can be considered as a model-checking problem, and hense studying alternative design in the presence of clients is possible at the time of system development. Policy Enforcer in our proposal is similar to the one discussed by Schneider [7], in which he proposes security automata, a variant of Buchi automata. Since it is encoded in EMM, Policy Enforcer is a finte-state automaton. Relation to model-checking problem has not been discussed so far.

In this paper, a simple Guiding System was used to illustrate the idea. The proposed method can be applied to any application system where policy rules are explicitly mentioned to enforce client behavior. Applying the method to such application systems is one possible area to pursue. Last, as a theoretical aspect of the method, future work includes automated discovery of ranking function rk for D_p from the analysis of counterexamples of $D_c \wedge S \not\models r$.

References

1. MOMENT Web Page, http://moment.dsic.upv.es/
2. Clarke, E.M., et al.: Model Checking. MIT Press, Cambridge (1999)
3. Clavel, M., et al.: All About Maude – A High-Performance Logical Framework. Springer, Heidelberg (2007)
4. Jackson, M.: Requirements & Specifications. Addison-Wesley, Reading (1995)
5. Jackson, M.: The Role of Formalism in Method. In: Wing, J.M., Woodcock, J.C.P., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, p. 56. Springer, Heidelberg (1999)
6. Kesten, Y., Pnueli, A.: Verification by Augmented Finitary Abstraction. Information and Computation 163(1), 203–243 (2000)
7. Schneider, F.B.: Enforceable Security Policies. Transactions on Information and System Security 3(1), 30–50 (2000)

A Model-Driven Software Development Approach Using OMG DDS for Wireless Sensor Networks

Kai Beckmann and Marcus Thoss

RheinMain University of Applied Sciences, Distributed Systems Lab,
Kurt-Schumacher-Ring 18, D-65197 Wiesbaden, Germany
{Kai-Oliver.Beckmann,Marcus.Thoss}@hs-rm.de

Abstract. The development of embedded systems challenges software engineers with timely delivery of optimised code that is both safe and resource-aware. Within this context, we focus on distributed systems with small, specialised node hardware, specifically, wireless sensor network (WSN) systems. Model-driven software development (MDSD) promises to reduce errors and efforts needed for complex software projects by automated code generation from abstract software models. We present an approach for MDSD based on the data-centric OMG middleware standard DDS. In this paper, we argue that the combination of DDS features and MDSD can successfully be applied to WSN systems, and we present the design of an appropriate approach, describing an architecture, metamodels and the design workflow. Finally, we present a prototypical implementation of our approach using a WSN-enabled DDS implementation and a set of modelling and transformation tools from the Eclipse Modeling Framework.

1 Introduction

Within the community developing complex distributed systems, the software crisis has been present for a long time already. For small-scale embedded systems, the situation is exacerbated by the fact that resources are scarce and thus code quality is rather defined in terms of compactness and optimisation for the specific platform than modularity and maintainability. This applies even more to software intended for cheap mass products with short life cycles.

As one possible escape from the crisis, model-driven software development (MDSD) has been proposed. MDSD promises code generation from abstract software models, which eliminates the error-prone manual design and implementation stages. As another important aspect for industrial mass production of embedded systems, MDSD can further help establishing product lines by managing line differences at an abstract, requirements-driven modelling level. In this paper, we will not further introduce the basics of MDSD; a good coverage of the topic can be found in [19].

MDSD has already found widespread use in business software scenarios like SOAs running on systems with relatively homogeneous architectures and comparatively abundant resources. Compared to those architectures, wireless sensor

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 95–106, 2010.

network (WSN) applications pose the additional challenges of close coupling of the code to the hardware platform including its peripherals, and a distinct lack of resources. Yet the prospect of being able to leverage MDSD for WSN environments is tempting, and in this paper we present an MDSD approach specifically designed to address those challenges.

WSNs heavily depend on the communication semantics used like publish-subscribe or event-based mechanisms, and on technical parameters like local node resources, transmission range, bandwidth and energy budgets. Regarding the communication aspects, we sought a solution whose semantics fit the requirements of sensor networks, that is, efficient data distribution, scalability and low overhead. We opt for the use of a data-centric middleware platform because we consider this paradigm a satisfactory match for WSN requirements, as described e.g. in [8]. Within this paper though, we will not elaborate on the pros and cons of communication semantics, but we state that we require that a data-centric approach be used for the application architecture targeted by our MDSD approach.

Instead of designing another data-centric middleware platform for our needs, we chose to build upon an existing standard, the Data Distribution Service for Real-time Systems (DDS) maintained by the Object Management Group (OMG) [11]. Choosing DDS is not only motivated by its being a standard, though. Besides being defined around a data-centric publish-subscribe paradigm, it offers a rich object-based API providing a base platform for application development. Unfortunately, although DDS is published as an OMG standard, it is not widely known (yet). Therefore, and because we tightly interweave DDS semantics and the metamodels used in our MDSD approach, we dedicate a section of this paper to describing DDS and its applicability to wireless sensor network platforms.

When describing our approach, we will use the terms "platform independent" and "platform specific" and the abbreviations PIM and PSM for the respective models to describe dependencies on hardware, operating systems and communications facilities.

The following sections present a review of publications related to our work and of the OMG DDS standard, followed by the description of our MDSD approach and the prototype created. Finally, we summarize the achievements, problems and prospective extensions.

2 Related Work

There are several recent research approaches that deal with different parts of the software development problem for WSNs. Using a middleware layer is an approved way to simplify application development for a WSN [14] [10]. Along these lines, various projects with different approaches and focuses have been published: Mires [18] is a message-passing middleware and applies the publish-subscribe paradigm to routing of data from data sources to data sinks. The provided API does not enforce a specific data model, and the definition of syntax and semantics of the messages are the responsibility of the application. Mires

uses TinyOS and the protocols provided by it. With TinyDDS [4], an approach for a DDS-based middleware for WSNs was presented. TinyDDS is heavily based on TinyOS and nesC, which is the only programming language supported. Consequently, TinyDDS adapts the DDS API, and to some extent the semantics as well, to the event-based TinyOS. Finally, in [5], a model-driven performance engineering framework is proposed, which relies on TinyDDS. Using a configuration for application, network and hardware related features, the resulting performance is estimated and can be optimised without the need for run-time observation or simulations.

Model-driven software development is a recent promising attempt to address the WSN software development problem. Most of the related work for WSNs concentrates on the modelling of applications with subsequent code generation. In [9], applications are designed with domain and platform specific models and a component-based architecture description. The models are transformed to a TinyOS- and nesC-based metamodel. This is also the targeted platform, and its functionality is used by the generated application. [15] proposes the usage of meta programming languages for the definition of domain specific languages (DSL) suitable for domain experts. Algorithms or applications expressed in the DSL are platform independent and can be used for simulations or translated to a specific platform. The ScatterClipse toolchain proposed in [1] puts the focus on a development and test cycle for WSN applications. It uses an extension of ScatterFactory [2], an infrastructure for developing and generating applications for the ScatterWeb [17] WSN hardware platform. Besides the applications, the test cases are modelled as well and the application, environments for deployment and test, and modules are generated. The selection of required functionality during the generation process is performed at library level. The combination of MDSD with an embedded middleware system is proposed in [6], where a component-based middleware is tailored according to a model expressed in a purpose-built DSL. The middleware thus described connects hardware or application containers within the WSN. Specifically, platform independence is achieved by modelling the container interfaces with the DSL.

The usage of tools and frameworks of the eclipse project has become common among MDSD projects within the context of WSNs. But, so far, few approaches have been proposed which are using MDSD to generate an application specific middleware for WSNs. For embedded systems, MDSD has been an accepted technology for some time already. An example for a generic approach is proposed in [21], where a component/container infrastructure and the underlying middleware are generated in a MDSD-based process. Several DSLs are used to model the different elements, like interfaces, components and the overall embedded system. The application logic must be inserted manually in the generated infrastructure using the modelled interfaces. Another commercial approach proposed in [16] uses the abstraction of process variables as a base layer for implementing platform independent embedded applications for the automation context. The actual, platform specific data sources and sinks for the process variables are configured separately. Glue code to connect the local hardware or remote devices

over different field buses can be generated in the process. Both approaches are using a dedicated interface for the application to access the generated infrastructure.

3 OMG's Data Distribution Service

DDS is an open standard for data-centric publish-subscribe middleware platforms with real-time capabilities published by the OMG [11]. The OMG is known for technologies like CORBA [12] and UML, and OMG efforts generally aim at portable, vendor-neutral standards. For DDS, this ensures that the scope of its semantics is not limited like ad-hoc solutions of research groups and single vendors. It can rather be assumed that DDS will remain an important player in the data-centric middleware landscape for a while.

Like CORBA, DDS uses an object-based approach to model DDS system entities, the API and communication data. At the heart of DDS data modelling, *topics* are used to denote possible targets of publications and subscriptions, like shown in Fig. 1. DDS topic types are themselves modelled using the OMG IDL type set, offering complex types like structures and arrays and simple type atoms like integers. For a given topic, data publications and subscriptions exchange data samples with corresponding types. At the topmost level, DDS defines *domains* for application-level grouping of data types and communication domains. Domains also model the data space and thus the naming and addressing of topics and subordinate elements within a domain.

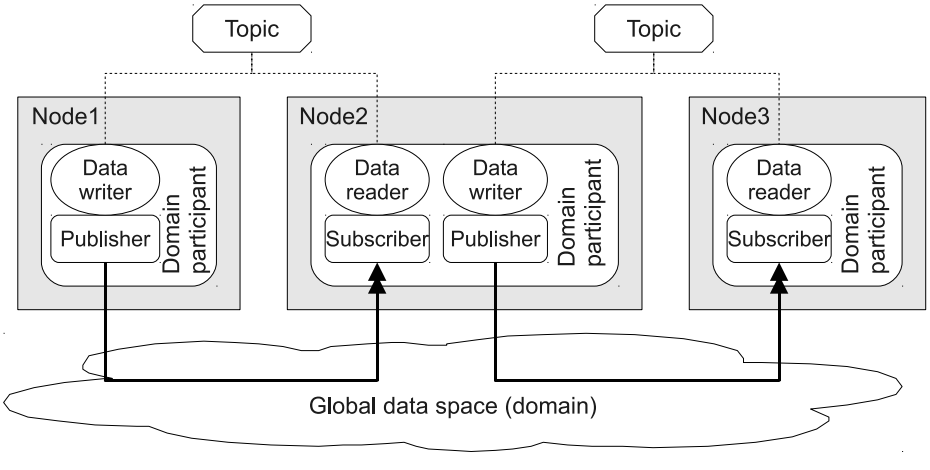


Fig. 1. Essential DDS architecture

Both the data model and the application interface of DDS are platform independent. They are specified in OMG IDL, for which language bindings exist that describe platform specific mappings. With these mappings, topic data and API

elements are mapped to types, variables and methods, expressed in a concrete implementation language. Thus, the use of DDS alone already offers a high level of abstraction for data modelling and use of the communications API by the application, both of which are major parts of the problem space.

The DDS standard also describes a rich set of QoS requirements and guarantees that can be claimed and offered by communication partners, respectively. Among the areas addressed by DDS QoS policies are reliability of communication, bandwidth consumption, and latencies.

For the distribution of topic data between publishers and subscribers, DDS assumes a datagram service offering message integrity, and routing and broadcast capabilities. There is a recommended wire protocol standard (RTPS) [13] published along with the DDS standard proper, which is supported by many DDS implementations.

The rich API and the design of RTPS suggest that DDS was not originally designed for small embedded wireless sensor nodes and networks with small packet sizes. To overcome the latter problem of RTPS, we also designed and realised an alternative DDS wire protocol (SNPS) optimised for WSN communications technology like ZigBee [3]. Additionally, subsetting of the DDS type space and API was necessary to achieve application footprints matching WSN node resources. Since this paper focuses on the MDSD aspects, we will not elaborate on those efforts here, but in [3] we were able to show that, on a technical level, DDS can be used in WSN scenarios. The general usability of the DDS approach for WSNs will be considered in the next section.

4 Approach

An MDSD approach generates executable applications from formal software models. For this, appropriate models to express the application and system design must be defined. This is done in turn using metamodels that describe the semantics and a syntax to define possible model elements. When an abstract syntax is chosen for the metamodels, one of a set of domain specific languages (DSLs) can be used to actually formulate a concrete model adhering to the metamodel.

In our approach, we further differentiate between platform independent models (PIMs) describing entities of the problem space and platform specific models (PSMs) targeting the solution space. With appropriate mappings, the transition from the PIM to the PSM domain can be automated such that human design activities can be restricted to the PIM level.

The main flow of MDSD activities is directed from PIM to PSM levels. As a refinement, although the ultimate output of a MDSD system should be application code, intermediate levels can be introduced that are already more specific in terms of platform specialisation (e.g. OS API) but have not yet reached the final code generation level. To restrict the impact of modifications of single model elements such that only the resulting model elements and code fragments affected must be re-generated, iterations within the design flow must be supported as well.

To summarize, the main building blocks of our approach are:

- a set of metamodels and DSLs describing the input models for the engineering process
- a pool of libraries and code fragments that constitute the composition elements and
- a tool chain consisting of parser and generator tools that process model information to assemble code pool elements to a deployable application

Although MDSD generally means to generate application code from abstract software models, the differences between business software scenarios and the architectures of WSNs must also be taken into account. For WSNs, an application must be regarded as a node-spanning entity because no single sensor node can achieve the goal of the application alone. Still, technical limitations concerning communication capabilities and node resources considerably influence the design of both the overall WSN structure and the software running on a single node. It is therefore desirable to capture the resources and non-functional requirements at the model level with the same priority as the semantic goals of the application.

From an MDSD output standpoint, WSNs require many instances of sensor node software to be built and deployed in order to form the overall application as the result of an orchestration of the application fragments. It might be tempting to initially assume that the code generated for all nodes can be identical, but the assumption of a strictly homogeneous hardware set-up and assignment of responsibilities limits the applicability of the approach. We therefore strive to support sets of nodes with heterogeneous capabilities.

4.1 Metamodels and Layers

To efficiently and completely capture the structure of a target application, we found that a single metamodel is not sufficient. Instead, we propose to diversify the metamodels available to the developer according to the different facets of the problem space to be solved. With this basic attitude, we could identify the following aspects to be modelled:

- problem domain-oriented typing of data to be exchanged
- topics as addressable entities for publications and subscriptions
- participation of a sensor node as publisher and subscriber within a domain
- QoS requirements for data exchange and node capabilities and
- hardware and operating system level configuration of a sensor node

Based on these requirements, we decided on four classes of metamodels to be available in our current MDSD approach: *data types*, *data space*, *node structure*, and *node config*. For the modelling of DDS *data types*, the PIM used is naturally based on OMG IDL [12], enriched with a topic type. Domains and the arrangement of topics within are independently modelled in a *data space* model, which can also capture QoS requirements to be fulfilled by the user of a topic. The roles of nodes as publishers and subscribers to topics are captured in *node*

structure models. Again, optional model elements can be used to express QoS requirements for the node entities modelled. Finally, *node config* models are used to describe capabilities and resource limits of the physical node hardware and the operating system possibly running on a node.

4.2 DDS Integration

We chose DDS as the underlying data-centric middleware system. Actually, the current design of our approach depends on a combination of features we could not find in any other middleware approach that is currently available. Besides being a likely choice for a data-centric middleware because it is an OMG-supported standard, DDS is specified in a platform independent manner, which facilitates its adoption at the PIM layer of a MDSD framework. Additionally, DDS supports QoS requirements that can be exploited to match resource limitations of sensor nodes and communication channels. The API is object-based, and for applications focusing on data exchange, most of the application code activities are covered by the DDS API. For code generation, this means that most of the code generated will consist of the implementation of the DDS API and its invocations.

The usage of DDS also enables a variation of the approach which skips the generation of application code at least initially. Instead, the generated DDS interfaces are used during run-time through dynamic invocation from possibly hand-crafted code.

4.3 Architecture

The architecture of the MDSD framework is shown in Fig. 2. From top to bottom, the degree of platform dependency increases. Input is given at the model layer which shows the four categories of DSL model descriptions that must be specified to define the application topic types, the DDS configuration describing communication relationships, and the system configuration at node level, respectively. Further input data from the left provides system-wide constraints, resource characteristics and pools of software fragments and is normally left unchanged across applications.

Descending from top to bottom, four lanes can be identified showing the metamodel and DSL elements participating in the parsing of the four model input categories into the still platform independent internal object representation. These object trees are subsequently checked for mutual compliance with QoS constraints defined in the model inputs, like resource limits in the configuration definition of a node. While still at the PIM level, optimisations based on information about the dependencies among DDS components can take place in the next step. This allows for re-use of DDS components within a single node.

The step from PIM to PSM is a combination of the transformation of application specific parts to program code, and the selection of DDS and system-level code fragments needed to support the application functionality. The actual program code to be deployed on a sensor node ultimately consists of the DDS

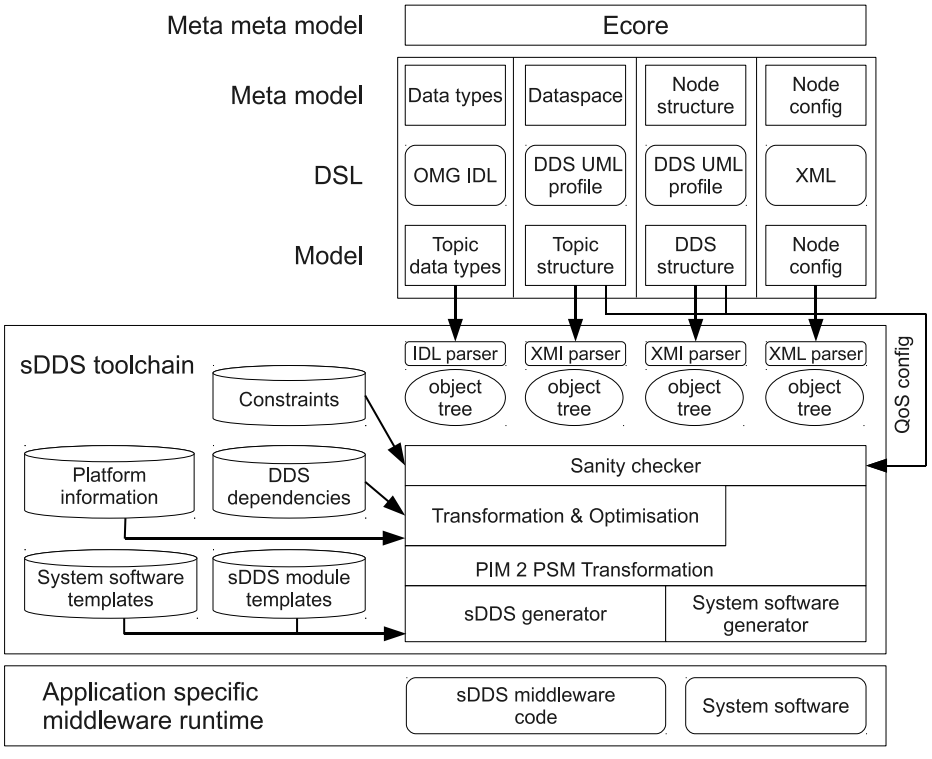


Fig. 2. Overall MDSD architecture

application that is composed in a linking step from those prefabricated code fragments and the generated code.

4.4 Workflow

From an application designer's point of view, interaction with the development workflow shown in Fig. 3 happens at the model creation and modification stage, which is iterated with parsing and sanity checking steps until consistency is reached within the PSM set. The subsequent optimisation and transformation steps are carried out automatically, eventually delivering deployable application code as output. Up to now, this is the end of our MDSD workflow chain, which leaves to the user the actual deployment, testing, and feedback of testing results into the design stages.

Many software engineering approaches rely on a cyclic model incorporating recurring phases in the application life-cycle. Because re-iteration of engineering steps is usually required after changes of requirements or fixing of design flaws, our approach strongly supports recurrent execution of single transformation and generation stages. The separation of model domains adds to the flexibility when

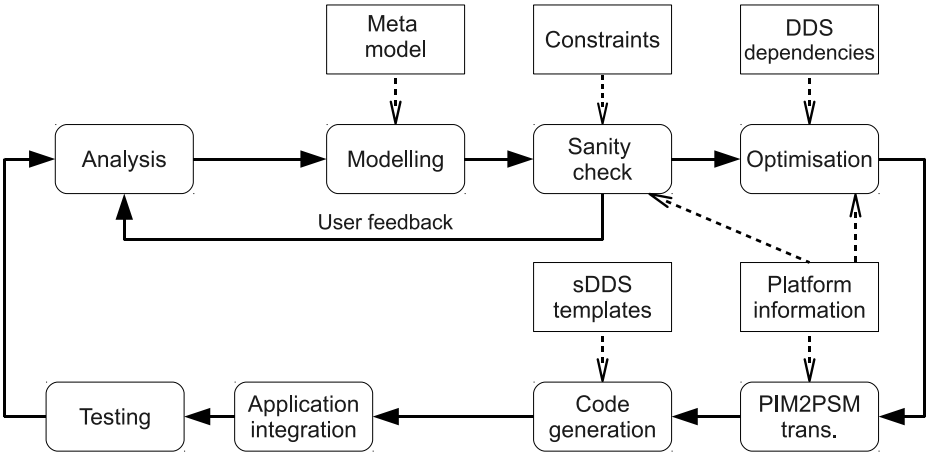


Fig. 3. MDSD workflow

adopting model changes because only the paths in the transformation tree containing changed models need to be considered. For monolithic and, thus, more closely coupled models, the impact of change is generally greater.

5 Prototype

The concepts described so far have been implemented in a prototype. For this, the following activities were necessary:

- selection of a target hardware and DDS environment for testbed applications
- selection and adaption of a set of tools for parsing and transformation of the models and for code generation
- definition of actual metamodels defining the PIMs
- development of code fragments serving as input for the code generator
- development of selectable code fragments for the DDS and OS level code repositories

Our implementation currently supports two target environments. For both, the DDS system (sDDS) and protocol implementation (SNPS) were developed in our laboratory [3]. To achieve fast turnaround times and allow for easy debugging and visualisation, COTS PCs and an Ethernet/IP/UDP-based protocol variant are used. For testing on an embedded target, we support an IEEE 802.15.4/ZigBee System-on-a-Chip platform (Texas Instruments CC2430) [20], [22] for the WSN nodes and versions of sDDS and SNPS matching the resource and packet size limits of the node devices. Of course, intimate knowledge of the implementation of a DDS system and access to the source code greatly facilitates the successful preparation of code fragments for the generator and the code repositories.

For the toolset, we chose the former openArchitectureWare tools Xtext for DSL definition, Xtend for model transformations, and Xpand for code generation, which are now part of the Eclipse Modeling Project [7].

The metamodels created basically adhere to the concept of four metamodel classes for data types, data space, node structure, and node configuration, as presented in the previous section. Up to now, for simplicity, we have only defined a single DSL that captures the features of all PIM variants. Deriving different DSLs from the metamodels can lead to greater flexibility and clarity in more complex implementations of our method, but the separate editing and processing of the PIM is independent from the number of DSLs used.

As a starting point for the creation of code fragments served a generic prototype application for our sensor nodes and our sDDS implementation. One of the advantages of using DDS is the inherent object structure, which naturally permits the separation of code fragments into object implementations. From the prototypical implementation, the structure of a generic sDDS application was derived; it served as a template for the implementation of the initialisation code generator.

Finally, a full-circle run of the prototypical tools in the development cycle delivered sDDS node application code implementing basic DDS data exchange functionality for the PC and CC2430 platforms, respectively, and the node applications could be deployed successfully. It could thus be shown that the inherent flexibility of the model-driven approach can be combined with the delivery of usable, compact output code. Especially the availability of only 128 KiB program memory on the CC2430 platform is honoured, and it should be noted that about 80% of the available code space is consumed by TI's ZigBee stack implementation, which drastically constrains the application's memory footprint allowance. For the basic functionality targeted here, the combined sDDS and application code footprint amounts to 25 KiB.

6 Summary and Future Work

Starting with the objective to facilitate software development for WSN architectures, we have presented an MDSD approach that relies on a set of meta models for differentiated requirements, and a communications and run-time infrastructure based on DDS.

The diversity of the metamodels used for specific facets of the requirements domain aids the expressiveness of the models created. Separate transformation of the models at the PIM level further facilitates the design of parser and transformation engines because the models are more specific, and thus, leaner. An intermediate, common PIM representation finally permits optimisation within the PIM space before the PIM to PSM transformation is applied.

DDS, that was chosen as a basis for the communication semantics and target application architecture, offers both a PIM representation for the modelling level and PSM mapping. The use of DDS also reduces the need for an additional rich OS API, and with the OMG it is supported by a major player of the professional IT community.

Using DDS for WSNs is a novelty, and we could not find any other approach combining MDSD with DDS and WSNs, but we could show that it is not only applicable but even beneficial if the DDS implementation and the transport protocol have been designed appropriately. Our implementation successfully targets a small-scale sensor node platform without sacrificing essential properties of the DDS core standard. This is an important aspect, since the DDS API can cater for the data processing design of a WSN application beyond mere data transport.

The basic design of our approach is defined at an abstract level, and we consider it to be sufficiently complete in its current state to be fully implemented without major changes to the basic ideas. Our implementation has not yet reached that coverage, though. The potential of diversifying the DSLs at PIM level has not been exploited, optimisation within PIM space is still rudimentary, and the set of available code fragments for the generation stages is far from exhaustive.

Furthermore, integrated support for the development of additional, possibly complex application code is not yet addressed by our approach. Here again, the usage of the PIM-level DDS API could facilitate an extension of the MDSD design to model the application logic based on an additional metamodel and DSL, and to generate tightly integrated, application specific middleware code.

We expect our MDSD implementation to grow with the completeness and the usage of the sDDS and SNPS implementations for WSNs in ongoing projects within our research group. Since SNPS is also available for IP/UDP environments, we hope to foster some interest in the general application of MDSD to DDS-based systems beyond the WSN scope.

References

1. Al Saad, M., Fehr, E., Kamenzky, N., Schiller, J.: ScatterClipse: A Model-Driven Tool-Chain for Developing, Testing, and Prototyping Wireless Sensor Networks. In: International Symposium on Parallel and Distributed Processing with Applications, pp. 871–885. IEEE, Los Alamitos (2008)
2. Al Saad, M., Fehr, E., Kamenzky, N., Schiller, J.: ScatterFactory2: An Architecture Centric Framework for New Generation ScatterWeb. In: 2nd International Conference on New Technologies, Mobility and Security, pp. 1–6. IEEE, Los Alamitos (2008)
3. Beckmann, K.: Konzeption einer leichtgewichtigen, datenzentrierten Middleware für Sensornetze und eine prototypische Realisierung für ZigBee. Master-thesis, RheinMain University of Applied Sciences (2010), http://wwwvs.cs.hs-rm.de/downloads/extern/pubs/thesis/beckmann10.pdf
4. Boonma, P., Suzuki, J.: Middleware Support for Pluggable Non-Functional Properties in Wireless Sensor Networks. In: IEEE Congress on Services - Part I, pp. 360–367. IEEE, Los Alamitos (2008)
5. Boonma, P., Suzuki, J.: Moppet: A Model-Driven Performance Engineering Framework for Wireless Sensor Networks. The Computer Journal (2010)
6. Buckl, C., Sommer, S., Scholz, A., Knoll, A., Kemper, A.: Generating a Tailored Middleware for Wireless Sensor Network Applications. In: IEEE International Conference on Sensor Networks, Ubiquitous and Trustworthy Computing, pp. 162–169. IEEE, Los Alamitos (2008)

7. Eclipse Modeling Project, http://www.eclipse.org/modeling/
8. Karl, H., Willig, A.: Protocols and Architectures for Wireless Sensor Networks. Wiley, Chichester (2007)
9. Losilla, F., Vicente-Chicote, C., Álvarez, B., Iborra, A., Sánchez, P.: Wireless sensor network application development: An architecture-centric MDE approach. In: Oquendo, F. (ed.) ECSA 2007. LNCS, vol. 4758, pp. 179–194. Springer, Heidelberg (2007)
10. Masri, W., Mammeri, Z.: Middleware for Wireless Sensor Networks: A Comparative Analysis. In: International Conference on Network and Parallel Computing Workshops, pp. 349–356. IEEE, Los Alamitos (2007)
11. Object Management Group: Data Distribution Service for Real-time Systems, Version 1.2 (2007)
12. Object Management Group: Common Object Request Broker Architecture (CORBA) Specification, Version 3.1 (2008)
13. Object Management Group: The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification, Version 2.1 (2009)
14. Römer, K., Kasten, O., Mattern, F.: Middleware challenges for wireless sensor networks. ACM SIGMOBILE Mobile Computing and Communications Review 6(4), 59–61 (2002)
15. Sadilek, D.A.: Prototyping and Simulating Domain-Specific Languages for Wireless Sensor Networks. In: ATEM '07: 4th International Workshop on Software Language Engineering (2007)
16. Schachner, R., Schuller, P.: Zusammenstecken per Mausklick, Sonderheft Automatisierung & Messtechnik. Markt&Technik, pp. 27–30 (2009)
17. Schiller, J., Liers, A., Ritter, H., Winter, R., Voigt, T.: ScatterWeb - Low Power Sensor Nodes and Energy Aware Routing. In: Proceedings of the 38th Annual Hawaii International Conference on System Sciences (2005)
18. Souto, E., Guimarães, G., Vasconcelos, G., Vieira, M., Rosa, N., Ferraz, C., Kelner, J.: Mires: a publish/subscribe middleware for sensor networks. Personal and Ubiquitous Computing 10(1), 37–44 (2005)
19. Stahl, T., Voelter, M., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. Wiley, Chichester (2006)
20. TI: CC2430 A True System-on-Chip solution for 2.4 GHz IEEE 802.15.4 / ZigBee (2008)
21. Voelter, M., Salzmann, C., Kircher, M.: Model Driven Software Development in the Context of Embedded Component Infrastructures. In: Atkinson, C., et al. (eds.) Component-Based Software Development for Embedded Systems. LNCS, vol. 3778, pp. 143–163. Springer, Heidelberg (2005)
22. ZigBee Alliance, http://www.zigbee.org/

Reactive Clock Synchronization for Wireless Sensor Networks with Asynchronous Wakeup Scheduling*,**

Sang Hoon Lee[1], Yunmook Nah[2], and Lynn Choi[1]

[1] School of Electrical Engineering,
Korea University, Seoul, Korea
{smile97,lchoi}@korea.ac.kr
[2] Department of Electronics and Computer Engineering
Dankook University, Yongin, Korea
ymnah@dku.edu

Abstract. Most of the existing clock synchronization algorithms for wireless sensor networks can be viewed as proactive clock synchronization since they require nodes to periodically synchronize their clock to a reference node regardless of whether they use time information or not. However, the proactive approach wastes unnecessary energy and bandwidth when nodes don't use time information for their operations. In this paper, we propose a new clock synchronization scheme called Reactive Clock Synchronization (RCS) that can be carried out on demand. The main idea is that a source node initiates a synchronization process in parallel with a data communication. To propagate clock information only when there is traffic, we embed the synchronization process in a data communication process. The results from detailed simulations confirm that RCS consumes only less than 1 percent of the energy consumption compared to two representative existing algorithms while it improves the clock accuracy by up to 75.8%.

Keywords: Clock synchronization, wireless sensor network, wakeup scheduling, media access control, preamble sampling.

1 Introduction

Time is an indispensable element of information processed by various branches of computer science such as operating systems, distributed systems, and communication networks. Especially, wireless sensor networks (WSNs) that are often used to monitor real-life environmental phenomena require time information to accurately measure external events or to coordinate various operations among the sensor nodes.

Coordinated universal time (UTC) is a time standard based on International Atomic Time (TAI) [1]. UTC is commonly used as a reference time for Internet. However, for wireless sensor networks that often assume GPS-free low-cost sensors,

* This work was supported by a Korea University Grant.
** This work was supported by Mid-career Researcher Program through NRF grant funded by the MEST (No. 2010-0011271).

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 107–118, 2010.

a sensor node may not have an access to the UTC. Instead, a node is often designated as a reference node and the reference time offered by this node is used as the standard time to validate the time information. The process of adjusting the clock of each individual node to the clock of a reference node is called clock synchronization.

Most of the existing WSN clock synchronization algorithms [2, 3, 4, 5, 6, 7] require sensor nodes to periodically synchronize their clock to a reference node to maintain the clock error under a certain threshold since they assume that any node can use time information at any time. We can classify these algorithms as proactive algorithms because all the nodes proactively synchronize their clock whether they use time information or not. Since the proactive approach can guarantee a certain level of clock accuracy all the time, it is good for networks where nodes frequently use time information for their operations. Among various operations on a sensor node, MAC operations are the most time sensitive since each node periodically listens to its neighbors to check for a possible communication.

In WSN a node usually employs periodic wakeup and sleep to reduce the energy consumption due to idle listening [8, 9, 10, 11, 12]. The interval and the duration of this wakeup must be scheduled and coordinated with other nodes for an effective communication. Existing wakeup scheduling techniques can be classified into two approaches: synchronous [10, 12] and asynchronous wakeup scheduling [8, 9, 11].

In synchronous wakeup scheduling the wakeup of each node is synchronized with the wakeup of its neighbors. Since nodes share their wakeup schedules and adjust them according to neighbor's wakeup schedule, nodes require a reference time to validate time information for wakeup scheduling. In other words, nodes always need to synchronize their clock to prevent the malfunction of MAC operation. Therefore, the proactive clock synchronization approach is a good choice for the synchronous wakeup scheduling.

In contrast, asynchronous wakeup scheduling allows each node to wake up independently. Since nodes don't share their wakeup schedules, nodes use only duration information to carry out MAC operations. In general, the duration error due to the frequency difference between two clocks isn't big enough to incur a malfunction. For example, the maximum relative error between two Intel PXA271's clocks is 60μs/s [13]. When a node sends a 200-byte packet at 20kbps, the duration error is 0.6μs which is smaller than a single bit transmission time of 50μs. Furthermore, this duration error is not accumulated as time goes by since the duration is valid for an operation. Therefore, the asynchronous MAC may work without the clock synchronization. However, WSNs commonly use time information at an application level to order the chronology of the sensing events or to compute the location information of an event [14]. Thus, sensor networks with the asynchronous wakeup scheduling still require a reference time to validate the time information. However, in WSNs with the asynchronous wakeup scheduling the proactive approach wastes unnecessary energy and bandwidth when there is no event to process.

For the event processing, time information is used by only a source node and a destination node. It means that only the two nodes need to exchange their clock information to synchronize their clocks. Therefore, for WSNs with asynchronous wakeup scheduling we don't need to synchronize the clocks of other nodes who do not participate in the communication.

In this paper, we propose a new clock synchronization scheme called RCS (reactive clock synchronization) that can be carried out in a demand-driven manner. The main idea is that we embed the clock synchronization process in a data communication process so that clock synchronization is performed only when there is traffic. RCS basically uses the offset and delay estimation algorithm [6]. The offset is the difference in the value of a clock from a reference clock. While a pair of nodes exchanges packets for a data communication, each node inserts a timestamp into each packet header. A sender calculates its clock offset to a receiver by using the timestamp. After the calculation, the sender adds the calculated offset to the offset received from the previous hop and delivers the new offset information to the receiver. Then, the receiver can get the accumulated offset from a source node to itself. Therefore, RCS requires three packet transmissions: two for calculating offset and one for delivering offset. By repeating this process hop by hop, the destination node can compute the clock offset from a source node to itself.

To evaluate the accuracy and the energy consumption of the proposed scheme, we perform detailed packet level simulations of RCS and two existing clock synchronization algorithms called TPSN [4] and FTSP [5]. The simulation results confirm that RCS consumes only less than 1% of the energy consumption compared to the existing algorithms when a network has light traffic. In addition, RCS improves the clock accuracy by 75.8% and 36.5% compared to FTSP and TPSN respectively. The reason why RCS can provide more accurate clock than the existing algorithms is that each source starts the clock synchronization right before delivering its message.

This paper is organized as follows. Section 2 introduces the background material for this paper: discussing the existing asynchronous wakeup scheduling algorithms for WSNs. Section 3 presents the proposed reactive clock synchronization algorithm. Section 4 comparatively evaluates the performance of the proposed algorithm using the detailed network simulations. Section 5 surveys the related works. Finally, section 6 concludes the paper.

2 Background: Asynchronous Wakeup Scheduling

In asynchronous wakeup scheduling each node independently wakes up. Since a sender cannot determine when a receiver will wake up, a sender sends a long preamble enough to cover the receiver's wakeup time. On a preamble reception the receiver further wakes up and both the sender and the receiver can participate for the communication. Depending on the packet exchange sequence, we can classify asynchronous wakeup scheduling algorithms into two approaches: a Preamble-Data-ACK (PDA) approach and a Preamble-ACK-Data (PAD) approach.

In the PDA approach, a preamble is a meaningless bit-stream whose only function is to make the receivers of the preamble to prepare for a data packet reception. All the nodes in the transmission range of a sender keep awake after receiving a preamble and prepare to receive a data packet. Only the receiver node which is named in the data packet header keeps receiving the data packet after receiving the header while other nodes go back to sleep. If the receiver successfully receives the data packet, it sends ACK to the sender. B-MAC [11] and WiseMAC [9] are the representative protocols of this approach. In B-MAC a preamble packet is long enough to cover all the

neighbors' wakeup time; the minimum length of a preamble should be equal to the duration of wakeup interval. In WiseMAC a receiver node piggybacks its wakeup schedule by using an ACK packet. Therefore, a sender can reduce the length of a preamble by computing the wakeup time of the receiver. Since a receiver inserts the remaining time until the next wakeup into an ACK, WiseMAC does not require clock synchronization for accurate instant time information.

To address the problem due to long preambles, the PAD approach uses multiple short preambles. In addition, by specifying the receiver address in the preamble, all the other receivers of the preamble except the receiver can go back to sleep immediately, avoiding the overhearing problem of unintended receivers in the PDA approach. The receiver node replies to the sender that it can receive a data packet by sending an ACK packet. Then, a sender transmits a data packet without an additional ACK. Since only a receiver keeps awake, other neighbor nodes of a sender can reduce the energy consumption due to overhearing. X-MAC [8] is the representative protocol of this approach. In X-MAC a sender sends a short preamble and waits an ACK packet for a pre-specified duration. If there is no ACK packet, a sender sends additional preamble until it receives an ACK packet.

3 Reactive Clock Synchronization

In this section we introduce Reactive Clock Synchronization (RCS) scheme. After introducing the main idea of RCS, we discuss how RCS can be implemented in two types of the existing asynchronous wakeup scheduling algorithms.

3.1 Main Idea

When a sensor node detects an event, it generates a report data. The data may contain time information such as event detection time. For a destination node to use the time information, the source node and the destination node must share a reference time. Without sharing a reference time which requires a global clock synchronization process, another option is that the destination node converts the time information in the data packet into its local time. If we can compute the offset between the clocks of the source and the destination, the destination node can convert the time information in the data packet provided by the sender to its local time. Intuitively, the offset between a source and a destination is same as the sum of all the offsets between the nodes on a path from a source to a destination. Fig. 1 shows an example of a data delivery. A node D has to know the offset from a node A to itself to use the time information in the data packet generated by a node A. The offset between A and D (O_{AD}) can be calculated as '$O_{AB} + O_{BC} + O_{CD}$'.

During data communication, each intermediate node accumulates an offset by adding its local offset to the received offset from the previous hop. By using this accumulated offset, a destination node can approximate the offset between a source and itself. Since the offset is calculated on demand during the data communication process, a destination node can use the up-to-date time information. Therefore, we can minimize synchronization error due to clock skew by reducing the gap between the point of the last synchronization time and the point of the time information usage.

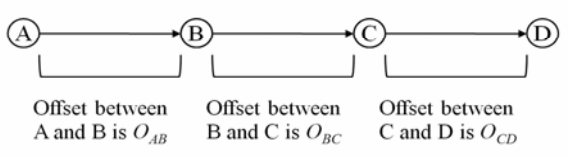


Fig. 1. An example of the accumulated offset

To implement the above distributed clock synchronization algorithm based on the accumulated offset value, we need three-way handshaking: request, reply, and offset delivery. A pair of nodes on a data delivery path carries out the offset-delay estimation algorithm through the request and the reply. After the calculation of an offset a sender delivers its offset value to a receiver. Since each operation requires only timestamps and node address, we can embed the information into the header field of an existing packet if possible.

3.2 In the PDA Approach

In this approach, if a node has a data packet to transmit, it first sends a preamble bit stream which indicates there will be a data packet transmission. We don't use the preamble for the offset-delay estimation since we need two-way message exchange and a sender will send a data packet after sending a preamble. Therefore, we will use a data packet and the following ACK packet to calculate the offset between a sender and a receiver. Since our clock synchronization process requires each node to send the calculated offset, we need an additional packet to deliver the offset. We call this additional packet as *clock packet*. Note that we may not guarantee the reliable transmission for the clock packet. On the clock packet transmission failure the receiver node may reuse the previous clock offset value. If this is the first synchronization between the sender and the receiver, the receiver approximates the clock offset by computing the difference between the send time and the reception time of the data packet, ignoring the transmission delay as in FTSP [5].

Fig. 2 shows an example of the clock synchronization process for the PDA approach where a node 1 is a source and a node 3 is a destination. In this example, to adjust the timestamp in the data packet, node 3 needs the clock offset from node 1. Assume that node 1 detects an event at T_0 and the local time of node 3 is T'_0 at that time. When node 1 sends a data packet, it inserts timestamp T_1 into the packet. Node 2 receives the packet at T_2 and sends an ACK packet at T_3. The ACK packet contains T_2 and T_3. After receiving the ACK at T_4, node 1 can estimate the offset and transmission delay from node 2 and sends a clock packet. Node 2 repeats the same process with the previous communication. After the offset delay estimation, node 2 can compute the offset from node 1 and the offset to node 3. Then, node 2 sends a clock packet which contains the sum of the two offset values. Then, node 3 can get the offset information between a node 1 and itself and adjust the time information in a data packet.

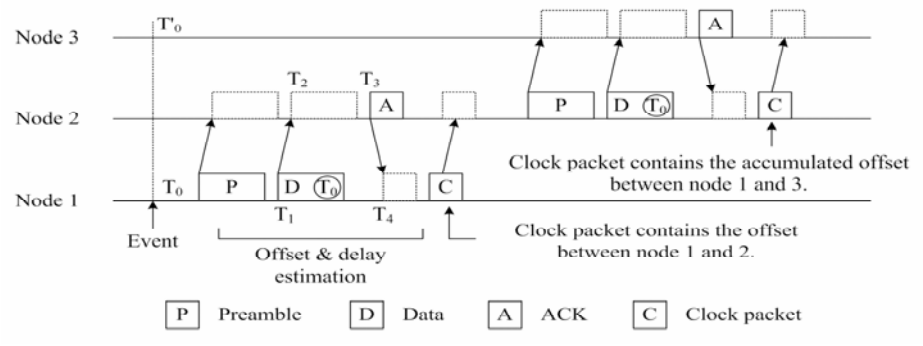


Fig. 2. An example of an offset estimation process in a PDA scheme

3.3 In the PAD Approach

In this approach we can use a preamble to carry out the offset and delay estimation process since a preamble contains additional information such as the destination address and an ACK packet will follow a preamble. We can carry out the offset-delay estimation while nodes exchange a preamble and an ACK packet. Therefore, we can insert the accumulated offset into the header field of a data packet. Different from the PDA approach, we need no additional packet to deliver the calculated offset.

Fig. 3 shows an example in this approach. Event detection scenario is the same as the previous example in Fig. 2. Although a sender can estimate the offset to a receiver before sending a data packet, it cannot modify the time information in a data packet at a MAC layer since a MAC protocol does not know the content of an application data. Therefore, a sender inserts the offset into the header field of a data packet.

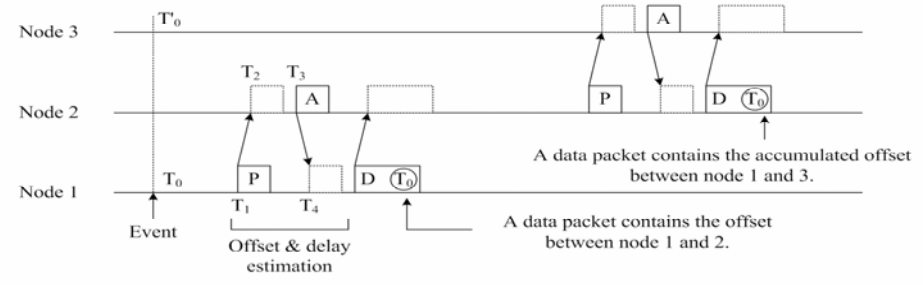


Fig. 3. An example of an offset estimation process in a PAD scheme

3.4 Clock Synchronization for Broadcast Data

In WSNs time information can be delivered not only through unicast data but also through broadcast data. For example, a sink can broadcast a query with time information throughout the network. We cannot employ the clock synchronization algorithm introduced in the previous section for a broadcast traffic since ACK packet is usually

not required for broadcast data. Thus, we employ the broadcast-based clock synchronization algorithm used in FTSP and S-MAC [12]. The broadcast-based clock synchronization doesn't try to measure the exact offset and delay. Instead, each node simply broadcasts its clock information and it assumes that it can estimate the message delay by using a certain probability function [5] or by assuming a pre-computed (often, zero) message delay [12].

Fig. 4 shows an example of the clock synchronization for a broadcast packet. In this example, node 2 assumes that T_2 is the same as T_1 and a node 3 assumes that T_3 is same as T_4. Therefore, the offset at node 2 and 3 are calculated as (T_2 - T_1) and (T_4 - T_3) respectively.

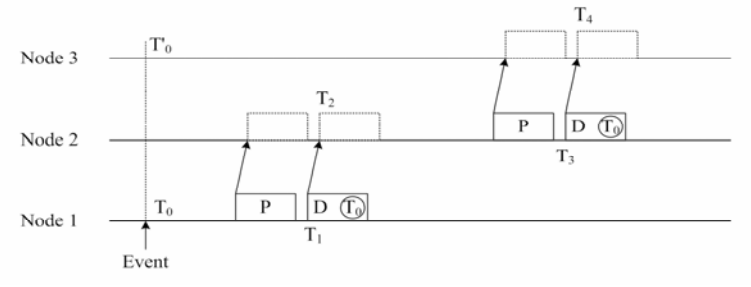


Fig. 4. An example of reactive clock synchronization for broadcast traffic

3.5 Expiration Time

After carrying out the offset-delay estimation, the calculated offset will be useful until the relative clock synchronization error is bigger than the predetermined threshold. If there is burst traffic, multiple data packets will be transmitted for a short time. Therefore, we can reuse the calculated offset for some time. This is called *expiration time*. Each node records the expiration time for every neighbor node after carrying out the offset-delay estimation. If there is another communication before the expiration time, a sender reuses the previous offset information. This can reduce the number of clock synchronization processes when there is burst traffic.

4 Simulation and Result

In this section we evaluate both the energy consumption and the accuracy of RCS by using detailed packet-level simulations. We chose TPSN and FTSP as reference schemes since they are the representative clock synchronization schemes for WSNs. TPSN is based on the offset and delay estimation algorithm while FTSP uses a broadcast-based synchronization. Since the original TPSN doesn't exploit the MAC-layer time-stamping proposed by FTSP, the clock accuracy of FTSP is generally higher than that of TPSN. However, in this simulation we assume the MAC-level time-stamping for both TPSN and RCS for a fair evaluation.

4.1 Simulation Methodology

We have implemented the detailed packet-level simulator using NS-2 [15] to model TPSN, FTSP and RCS. The parameters used in the simulations are summarized in Table 1. We use a grid topology with 400 nodes and a sink locates at the center of the network. We select a random source which generates 10 messages at once, each of which is 50 bytes long. Under each traffic condition, the test is independently carried out 10 times. For this simulation we use B-MAC and X-MAC as underlying MAC protocols. B-MAC is one of the most representative PDA asynchronous MAC protocols while X-MAC is one of the representative PAD MAC protocols.

We use two metrics to analyze the performance of RCS: the average per-node dissipated energy for the synchronization process and the average clock accuracy. The average per-node dissipated energy measures the total energy consumed for each node to carry out the clock synchronization. This metric doesn't include the energy consumption for the idle listening and the data communication. The metric indicates only the overhead required for the clock synchronization. The average clock accuracy measures the difference between the calculated time and the reference time.

Table 1. Parameters for the simulations

Simulation parameters	Value
Cycle time of a node	2 seconds
Time for periodic wakeup	1 ms
Power consumption for the transmission and reception	tx: 30mW, rx: 15mW
Power consumption for the idle state	15mW
Size of control packets used for clock synchronization	10 bytes
Clock synchronization period	10 minutes
Simulation time	1 hour
Number of nodes	400
Number of packets for each message	10
Maximum clock skew	50 ppm

4.2 Average Dissipated Energy

Fig. 5 compares the average per-node energy consumption of each scheme. As shown in Fig. 5 (a) the proactive algorithms consumes much more energy than RCS. RCS consumes only 0.004% energy of TPSN for clock synchronization on a PDA scheme since only those nodes which participate in the data communication carry out the clock synchronization. In contrast, both TPSN and FTSP require every node to participate in the clock synchronization. Therefore, RCS can substantially reduce the energy consumption overhead of the existing clock synchronization algorithms.

Fig. 5 (b) shows that RCS with expiration timer can substantially decrease the energy consumption overhead by suppressing unnecessary clock synchronization. Since a source node sends 10 continuous packets for each message, a source node carries out RCS only once when it sends the first packet of the message. If the time between two messages delivery is less than the expiration time, a source node does not have to carry out RCS when it sends the second message. However, if there is light traffic with small messages, such as a single packet message, the performance gain from the

expiration time becomes small. We also find that RCS on the PAD approach consumes 43% more energy than PDA approach since a sender sends multiple short preambles, which results in transmitting duplicated timestamps.

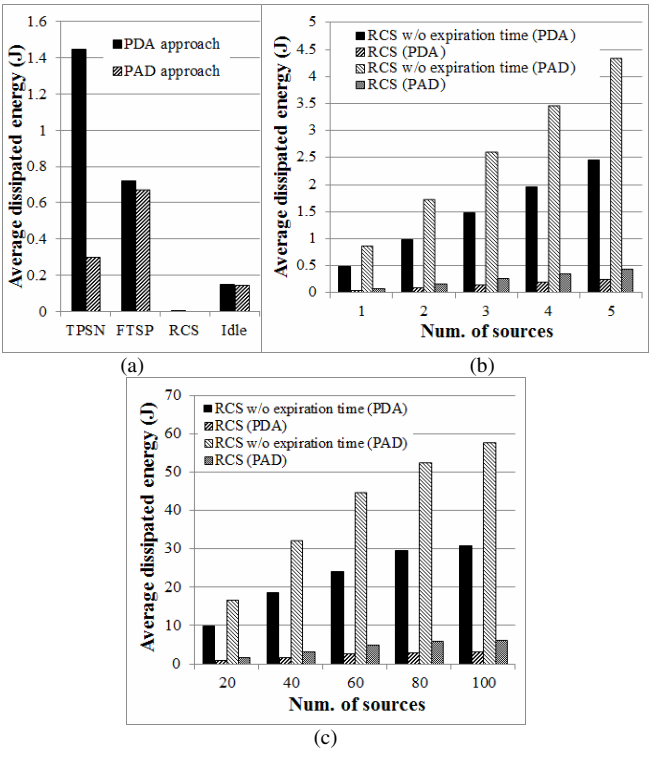


Fig. 5. Average per-node dissipated energy

Fig. 5 (c) also shows that the energy consumption of RCS is proportional to the number of sources. Since the nodes on the path between a source and a destination need to calculate the accumulated offset, the number of nodes which carry out the clock synchronization is proportional to the number of sources. However, as shown in Fig. 5 (b), the rate of energy consumption has slowed as the number of sources increases since different messages can be delivered through the same path. RCS with 100 sources consumes only 3.2 times more energy than RCS with 20 sources. When there are 100 sources, 62.2% of the forwarding nodes which participate in message deliveries can forward two or more messages.

4.3 Average Accuracy

Fig. 6 compares the average clock error of each scheme as we increase the hop distance from a source node. Since FTSP cannot exactly measure the message delay, the average clock error of FTSP is higher than that of TPSN which can compensate the error due to both clock offset and message delay. According to Fig. 8 (a), RCS

improves the clock accuracy by 75.8% and 36.5% compared to FTSP and TPSN respectively. The reason why RCS provides a more accurate clock than other schemes is that it carries out the synchronization process right before a destination uses the time information. In contrast to RCS, the existing schemes carry out the process in advance of the use of time information. Therefore, the increased clock skew adds to the clock error as time goes by since the last synchronization.

As shown in Fig. 6 (b), the average accuracy of two versions of RCS has almost the same performance. Since RCS without the expiration time carries out the synchronization process whenever a node sends a data packet, the accuracy of the full version of RCS is slightly higher. However, the expiration time can effectively reduce the energy consumption for the clock synchronization while RCS provides comparable clock accuracy. However, if we average clock error of all the nodes in a network, RCS will have higher total average clock error than the existing algorithms since only the nodes who participate in data delivery carry out clock synchronization process. The total average clock error of RCS including all the nodes in a network was 1.35ms that is 78 times higher than TPSN. Although the total average clock error of RCS is higher than others, it is no consideration since time information of un-synchronized nodes doesn't affect data processing on a sink.

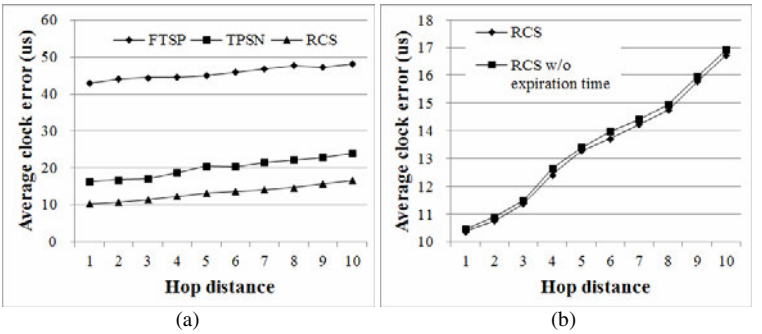


Fig. 6. Average clock error as we increase the hop distance

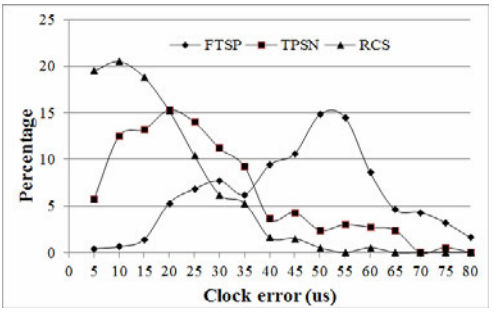


Fig. 7. Distribution of clock error between a pair of nodes

Fig. 7 shows the distribution of clock error from a source to a sink. As expected, the clock error distribution of RCS is closer to zero than others. The maximum clock error of RCS was 58μs while that of TPSN and FTSP were 72μs and 77μs. In RCS 50% of source nodes have clock error under 13μs while TPSN and FTSP marked 21 μsand 47μs, respectively.

5 Related Works

Most of the existing clock synchronization schemes proposed for WSNs can be classified as proactive schemes since they perform synchronization process periodically. To improve the accuracy of a clock, most of the existing algorithms often employ the offset-delay estimation algorithm since it can compensate error due to both message delay and clock offset.

Reference Broadcast Synchronization (RBS) [3] tries to improve the accuracy of traditional clock synchronization schemes such as Remote Clock Reading [2] and NTP [6]. In RBS, a reference node broadcasts a reference packet that contains no explicit timestamp. Then, recipients use the packet's arrival time as a reference point for synchronizing their clocks. By comparing the reception time of each receiver, RBS can remove send time and access time from the uncertain message delay factors. However, the message exchanges among receiver nodes cause an exponential increase in the number of synchronization messages. This also increases the energy consumption of RBS exponentially as the number of nodes in the network increases.

To address the scalability issue with RBS, the authors of TPSN [4] aimed at providing a scalable energy-efficient algorithm for WSNs. Similar with NTP, TPSN adopts a two-way message exchange approach to measure the clock offset and message delay. Each node synchronizes with its upper node in the tree hierarchy which is found by flooding a level-discovery packet. However, TPSN does not consider error due to clock skew.

Tiny/Mini-Sync [7] is also based on the two-way message exchange approach. Different from TPSN, this protocol considers the error due to clock skew in addition to the error due to clock offset and delay. To estimate clock skew, Tiny/Mini-Sync uses a history of clock information used in the past synchronization processes. By compensating both clock offset and clock skew, Tiny/Mini-Sync can carry out synchronization more accurately than TPSN.

While conventional synchronization algorithms require a message exchange between a pair of nodes, FTSP [5] adopts a flooding-based approach to further reduce the energy consumption needed for clock synchronization. By using the unidirectional broadcast, FTSP requires only a single clock synchronization message per node instead of a message exchange. In addition, it eliminates timestamp uncertainty by MAC layer time-stamping. However, the flooding scheme of FTSP causes unexpected collision and useless packet transmissions.

6 Conclusion

This paper proposes a new clock synchronization scheme called RCS, which is the first on-demand clock synchronization algorithm. RCS is fundamentally different

from the existing clock synchronization schemes in that a source node initiates the clock synchronization process only when there is traffic. This eliminates the overhead for periodic clock synchronization process. In addition, with the expiration timer RCS can further reduce the energy consumption required for the clock synchronization when there is burst traffic. RCS allows us to achieve much lower energy consumption for clock synchronization while preserving a comparable accuracy. The results from our detailed simulations suggest that RCS is very effective in reducing the energy consumption when the network has light or burst traffic, which is the case for WSNs.

References

1. International Bureau of Weights and Measures, http://www.bipm.org/en/home/
2. Cristian, F.: Probabilistic Clock Synchronization. Distributed Computing 3, 146–158 (1989)
3. Elson, J., Girod, L., Estrin, D.: Fine-Grained Network Time Synchronization using Reference Broadcasts. In: OSDI 2002, vol. 36, pp. 147–163 (2002)
4. Ganeriwal, S., Kumar, R., Srivastava, M.B.: Timing-sync Protocol for Sensor Networks. In: International Conference on Embedded Networked Sensor Systems, pp. 138–149 (2003)
5. Maróti, M., Kusy, B., Simon, G., Lédeczi, Á.: The Flooding Time Synchronizatoin Protocol. In: SenSys 2004 , pp. 39–49 (2004)
6. Mills, D.L.: Internet Time Synchronization: the Network Time Protocol. Transactions on Communications 39(10), 1482–1493 (1991)
7. Sichitiu, M.L., Veerarittiphan, C.: Simple, Accurate Time Synchronization for Wireless Sensor Networks. In: WCNC 2003, pp. 1266–1273 (2003)
8. Buettner, M., Yee, G.V., Anderson, E., Han, R.: X-MAC: A Short Preamble MAC Protocol for Duty-Cycled Wireless Sensor Networks. In: SenSys 2006, pp. 307–320 (2006)
9. El-Hoiydi, A., Decotignie, J.-D.: WiseMAC: An Ultra Low Power MAC Protocol for Multi-hop Wireless Sensor Networks. In: Nikoletseas, S.E., Rolim, J.D.P. (eds.) ALGOSENSORS 2004. LNCS, vol. 3121, pp. 18–31. Springer, Heidelberg (2004)
10. Lu, G., Krishnamachari, B., Raghavendra, C.S.: An Adaptive Energy-Efficient and Low-Latency MAC for Data Gathering in Wireless Sensor Networks. Parallel and Distributed Processing, 224 (2004)
11. Polastre, J., Hill, J., Culler, D.: Versatile Low Power Media Access for Wireless Sensor Networks. In: SenSys 2004, pp. 95–107 (2004)
12. Ye, W., Heidemann, J.S., Estrin, D.: Medium Access Control with Coordinated Adaptive Sleeping for Wireless Sensor Networks. IEEE Transactions on Networking 12(3), 493–506 (2004)
13. Intel XScale processor, http://www.intel.com/design/intelxscale/
14. Manley, E.D., Nahas, H.A., Deogun, J.S.: Localization and Tracking in Sensor Systems. In: SUTC 2006, pp. 237–242 (2006)
15. The Network Simulator ns-2, http://www.isi.edu/nsman/ns

On the Schedulability Analysis for Dynamic QoS Management in Distributed Embedded Systems

Luís Almeida[1], Ricardo Marau[1], Karthik Lakshmanan[2], and Raj Rajkumar[2]

[1] IEETA / IT - DEEC - Faculdade de Engenharia
University of Porto, Porto, Portugal
{lda,marau}@fe.up.pt
[2] Real-Time and Multimedia Systems Lab
Carnegie Mellon University, Pittsburgh, PA, USA
{klakshma,raj}@ece.cmu.edu

Abstract. Dynamic Quality-of-Service (QoS) management has been shown to be an effective way to make an efficient use of systems resources, such as computing, communication or energy. This is particularly important in resource-contrained embedded systems, such as vehicles, multimedia devices, etc.. Deploying dynamic QoS management requires using an appropriate schedulability test that is fast enough and ensures continued schedulability while the system adapts its configuration. In this paper we consider four utilization-based tests with release jitter, a particularly relevant feature in distributed systems, three of which were recently proposed and one is added in this work. We carry out an extensive comparison using random task sets to characterize their relative merits and we show a case study where multiple video streams are dynamically managed in a fictitious automotive application using such schedulability bounds.

Keywords: Real-time scheduling, Quality-of-Service, distributed embedded systems.

1 Introduction

Utilization-based schedulability tests are known since the 1970s [9] and they are extremely simple to compute given their linear time complexity. When used on-line in the scope of admission controllers of open systems, their time complexity can even be made constant by keeping track of the total utilization of the currently running tasks. However, this simplicity comes at a cost, which is a lower accuracy than other more complex schedulability tests, such as those based on response time or processor demand analysis. This disadvantage grows as the task model considers more aspects that are typical in real applications such as blocking, deadlines different from periods, offsets, non-preemption and arbitrary priority assignment policies [4].

Furthermore, recent developments in response time analysis brought to light new linear methods that are faster to compute, despite a small loss in accuracy

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 119–130, 2010.

[7] [3]. These tests are still more complex to evaluate than utilization-based tests but the difference became relatively small and their accuracy is still better.

Nevertheless, utilization-based tests are still the most adequate solution when it is important to manage utilization, for example in the scope of dynamic QoS (bandwidth) management, with tasks that allow variable worst-case execution times by switching between different algorithms or that can be executed at different rates. Therefore, those tests are an effective way to improve the efficiency in the use of system resources when the task set in the system can change on-line, either due to admission/removal of tasks or because the tasks exhibit multiple modes of operation.

In such scope, utililzation-based tests are the natural choice because they handle bandwidth directly. Thus, for example, when a task leaves the system, the bandwidth that is freed can be assigned to the remaining ones, according to an adequate policy, from fixed priority/importance, to elastic, weighted, even, etc. [5] [8]. This assignment cannot be done in a comparable manner using other kind of tests and it must be done in a more costly trial and error fashion.

When applying dynamic bandwidth management to distributed real-time systems there is one additional difficulty, which is the frequent occurrence of release jitter, both in the tasks and messages they use to communicate. In fact, messages are frequently released by tasks that do not execute periodically because of variable interference of higher priority ones in the processor where they reside and, similarly, tasks are frequently triggered by the reception of messages that do not arrive periodically for similar reasons [11].

The incorporation of release jitter in utilization-based tests was not done until very recently. In this paper we revisit the existing work on utilization-based schedulability tests that account for release jitter, we propose an enhancement to a previous test and we compare the performance of such tests with random task sets. Then we show a case study concerning a video system for driver assistance in a fictitious automotive application that illustrates the benefits of using such a schedulability test for dynamic management of communications in an Ethernet switch.

2 Related Work

Despite all the work on utilization-based schedulability tests for periodic task models carried out in the past four decades, to the best of the authors' knowledge, there has never been an extension to include release jitter until very recently. In fact, the first published result in that direction seems to have been by Davis and Burns in 2008 [6] where they showed a pseudo-utilization-based schedulability test that accounts for release jitter based on the results in [1]. In fact, it is shown that the release jitter that affects each task can be subtracted from its period, or deadline if shorter than the period, and used directly in the denominator of the respective utilization term. The total sum gives a pseudo-utilization value that can be compared against the Liu and Layland bounds. The same work has an extensive comparison among different tests, focusing on the relative performance of response-time based tests.

On the other hand, the authors have previously developed a different analysis applicable to both Rate-Monotonic and Earliest Deadline First scheduling that accounts for release jitter as an extra task [10]. This analysis results in a set of n conditions for n tasks. However, the authors also showed how such conditions could be reduced to just one single test that upper bounds the n conditions.

The comparison between the n conditions test and the test in [6] is not obvious, as it will be shown further on, since none dominates the other in all situations. Nevertheless, the latter does not directly reflect bandwidth, which makes it harder to use within the scope of dynamic bandwidth management schemes. Moreover, the comparison becomes even less obvious given the different underlying priority assignment policies. In fact, while the former assumes usual *rate-monotonic* priorities, the latter assumes a *'deadline minus jitter'-monotonic* policy. Naturally, either test can be optimistic whenever used with a priority assignment different from the assumed one.

In this paper we propose a different simplification of the n conditions test proposed in [10] that also results in a single bandwidth-based condition but which is more accurate than the one presented therein at the cost of additional overhead. The remainder of the paper will focus on a comparison of the presented tests.

3 Task Model and Previous Analysis

We consider a set Γ of n periodic tasks $\tau_i (i = 1..n)$, with period T_i and worst-case execution time C_i, which may suffer a jittered release of at most J_i. The deadlines are currently considered to be equal to the respective periods. We consider the tasks in the set to be sorted by growing period, except when explicitly referred. In the context of Rate-Monotonic scheduling (RM) τ_1 will be the highest priority task and in the context of Earlist Deadline First scheduling (EDF) τ_1 will be the task with the highest preemption level. For the moment we also consider tasks to be fully preemptive and independent.

The test in [6] copes with deadlines equal to or less than the periods and blocking but, for the sake of comparison with the other tests, we will constrain it to our model. In this case we will assume the task set sorted by growing 'period minus jitter'. The test then states that schedulability of the task set is guaranteed if condition (1) is satisfied. The test in [6] considers fixed priorities, only, but the same reasoning expressed therein allows to infer that it should also be applicable to EDF scheduling. Thus, we will also consider it in such case. $U^{lub}_{RM,EDF}(n)$ refers to the least upper bound on utilization for RM scheduling with n tasks and for EDF, i.e., $n(2^{\frac{1}{n}} - 1)$ and 1, respectively.

$$\sum_{i=1} \frac{C_i}{T_i - J_i} \leq U^{lub}_{RM,EDF}(n) \tag{1}$$

On the other hand, the test in [10] states that if the n conditions in (2) are satisfied, then the task set is schedulable.

$$\forall_{i=1..n} \sum_{j=1}^{i} \frac{C_j}{T_j} + \frac{\max_{j=1..i} J_j}{T_i} \leq U^{lub}_{RM,EDF}(i) \tag{2}$$

The work in [10] also provides a simplified test based on a single condition as in (3).

$$\sum_{i=1}^{n} \frac{C_i}{T_i} + \frac{\max_{i=1..n} J_i}{T_1} \leq U^{lub}_{RM}(n) \tag{3}$$

This condition may become very pessimistic in cases with a wide dispersion of tasks periods. In fact, a task with a longer period can accommodate a longer release jitter, leading to a term $\max_{i=1..n} J_i$ that can be close to, or longer than, T_1. This situation can be improved using a new simplified test with a single condition (4) that is more accurate than (3) despite more costly to compute.

$$\sum_{i=1}^{n} \frac{C_i}{T_i} + \max_{i=1..n} \frac{\max_{j=1..i} J_j}{T_i} \leq U^{lub}_{RM,EDF}(n) \tag{4}$$

It can be trivially verified that if condition (4) holds then all n conditions in (2) will also hold and thus this test also implies the schedulability of the task set. Note, also, that the term $\max_{i=1..k} J_i$ is now divided by $T(k)$ that grows with k and thus the potentially longer jitter of slower tasks does not have such a strong effect as in condition (3).

4 Comparing the Tests

In this section we compare the four conditions presented above in terms of schedulability accuracy, i.e., level of pessimism, and execution time. For both cases, we include one accurate analysis as a reference, namely a response time test for RM as in (5) [2] and a CPU demand test for EDF based on the analysis presented in [11] as in (6). For the sake of presentation, we will also refer to the test of condition (1) as RM whenever applied to fixed priorities, although these being assigned in 'deadline minus jitter' order.

$$\forall_{i=1..n} Rwc_i \leq T_i \textbf{ with } Rwc_i = R_i + J_i \textbf{ and } R_i = \sum_{j=1}^{i-1} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil * C_j + C_i \tag{5}$$

$$\forall_{t \in \Psi} h(t) \leq t \textbf{ with } h(t) = \sum_{i=1..n} \left\lfloor 1 + \frac{t - (T_i - J_i)}{T_i} \right\rfloor * C_i \tag{6}$$

$$\textbf{and } \Psi = \{t \in [0, L] \wedge t = m * T_i + (T_i - J_i), \forall i = 1..n, m = 0, 1, 2...\}$$

$$\textbf{with } L = \sum_{i=1..n} \left\lceil \frac{L + J_i}{T_i} \right\rceil * C_i$$

For the comparison we use synthetic task sets with random tasks. The generation method is the following. We start by generating a target global utilization (U), then we generate random tasks one by one with a period (T_i) uniformly distributed within $[1, 10]$ and utilization factor (U_i) within $(0, 0.2]$. The utilization of the last task is adjusted if the total utilization is more than 1% above the desired target U. Then, we compute the respective execution times ($C_i = T_i * U_i$) and we generate the values of release jitter (J_i) randomly with uniform distribution.

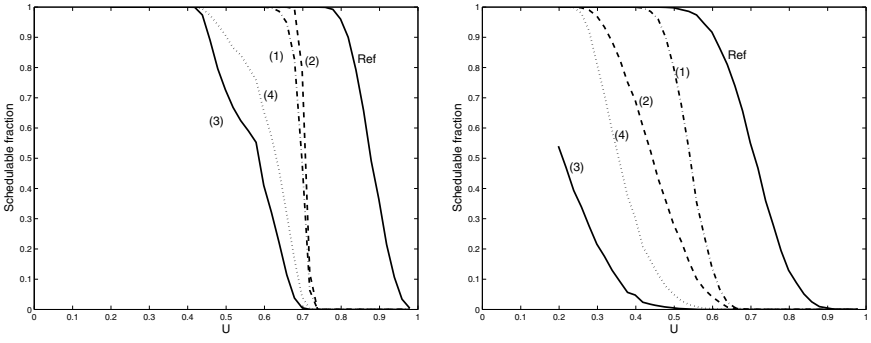
The values of U go from 0.2 to 0.98 in steps of 0.02 and for each particular point we generate 5000 random task sets. We gerenated two sets of experiments with different patterns of jitter. In one case we considered the same interval in the generation of jitter for all tasks, namely randomly distributed within $(0, 0.3]$. This is a relatively small jitter that impacts mainly the tasks with shorter periods. We call this the *flat* jitter profile. On the other hand, we generated another profile with a stochastically growing value of jitter according to the taks periods. Basically, the jitter J_i is randomly generated witin $(0, 0.5 * T_i]$. This means that longer tasks can suffer longer release jitter. We call this, the *linear* jitter profile.

The simulation experiments were carried out using Matlab on a common laptop with a Centrino CPU operating $1.2GHz$ and running the Windows_{XP} operating system. The execution times of the analysis that are shown next were not optimized and serve mainly for relative assessment.

Figure 1a shows the results obtained using the RM scheduling and the flat jitter profile. The rightmost curve shows the ratio of task sets that meet the response time test in (5) with both 'deadline minus jitter' priorities and rate-monotonic priorities. In this case, given the low amount of jitter, its impact in the priority assignment is minimal and, consequently, the response time test delivers very similar results with both policies (Ref1/2). This reference test assures that all task sets with this jitter profile and utilization roughly up to 74% are schedulable. Then we track how many of the task sets found schedulable by the reference test were also deemed schedulable by tests (1) to (4). This gives us a good measure of the level of pessimism embedded in these tests.

The values obtained for RM scheduling with the flat jitter profile are shown in Table 1. Test (2) performed best in this case, finding 75% of the sets considered schedulable by the reference test. Test (1) performed very closely, 73%. Then come tests (4) and (3) fiding 62% and 55% of the reference schedulable sets, respectively. It is expected that these two tests perform poorer than test (2) since they are a simplification of it and it is also expected that both tests perform similarly since with the flat jitter profile there is a non-negligible probability that the maximum jitter will affect the task with the shortest period, which leads to a similar result with both tests. Table 1 also shows the average and maximum time taken by each analysis with each set.

Interestingly, the situation changes substantially when we use the linear jitter profile, showing the expected strong impact of release jitter. In this case, the tasks with longer period may be affected by a longer release jitter in absolute terms, particularly, longer than the shorter periods in the system. Therefore, the imapct of the term $\max J$ in either tests (2), (3) and (4) can be significant,



(a) with the flat jitter profile. (b) with the linear jitter profile.

Fig. 1. Performance of different tests under RM

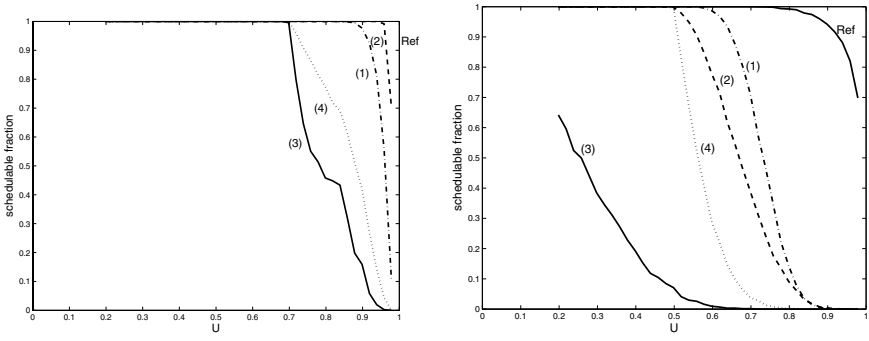
Table 1. Summary of results with the flat jitter profile under RM

	Ref 1/2	Test (1)	Test (2)	Test (3)	Test (4)
% of schedulable task sets found by the U-based tests	100%	73%	75%	55%	62%
analysis exec time (avg)	14.8ms	1ms	9ms	1.4ms	4.7ms
analysis exec time (max)	96ms	47ms	79ms	47ms	64ms

specially in test (3) in which such term is just divided by the shortest period. Consequently, this test becomes useless in practice whenever there are large release jitters in the system when compared to the shortest period. Conversely, the new test (4) that we present in this paper achieves a performance inferior but closer to the original n conditions test (2), since the stochastically growing terms of jitter are also divided by growing periods. The best performance with this jitter profile was, however, achieved with test (1).

The results can be observed in Figure 1b and in Table 2. Note that in this case, the larger jitter already causes a very slight difference in the reference test depending on whether it uses 'deadline minus jitter' (Ref1) or rate-monotonic priorities (Ref2). Such difference favours the former case given the optimality of that priority assignment in the presence of jitter [1]. In the Table, the results of test (1) are shown with respect to those of reference test Ref1 and those of tests (2) to (4) with respect to reference test Ref2.

The results for EDF scheduling are shown in Figures 2a and 2b for the flat jitter and linear jitter profiles, respectively. In this case, just one reference test was used (Ref) since the test in 6 is independent of the order in which the tasks are considered. The numerical results for both cases are shown in Tables 3 and 4. Basically, we achieved very similar results to those of the RM scheduling case but with a shift to the right due to the higher schedulability capacity of EDF, thus with an increment in the percentages of schedulable configurations found. Anyway, the relative performances of the diverse tests are similar to those achieved with RM.



(a) with the flat jitter profile. (b) with the linear jitter profile.

Fig. 2. Performance of different tests under EDF

Table 2. Summary of results with the linear jitter profile under RM

	Ref1 & Ref2	Test (1)	Test (2)	Test (3)	Test (4)
% of schedulable task sets found by the U-based tests	100%	68%	50%	11%	34%
analysis exec time (avg)	14.1ms	0.9ms	6.8ms	1.5ms	4.3ms
analysis exec time (max)	94ms	47ms	78ms	47ms	48ms

Table 3. Summary of results with the flat jitter profile under EDF

	Reference	Test (1)	Test (2)	Test (3)	Test (4)
% of schedulable task sets found by the U-based tests	100%	96%	99%	77%	84%
analysis exec time (avg)	47ms	1ms	8.8ms	1.5ms	4.9ms
analysis exec time (max)	159ms	47ms	79ms	47ms	62ms

Table 4. Summary of results with the linear jitter profile under EDF

	Reference	Test (1)	Test (2)	Test (3)	Test (4)
% of schedulable task sets found by the U-based tests	100%	69%	62%	13%	49%
analysis exec time (avg)	76ms	1ms	8.3ms	1.5ms	4.7ms
analysis exec time (max)	204ms	47ms	110ms	47ms	62ms

As a concluding remark, we can say that test (1) seems to perform better in the general case, only losing to test (2) with relatively low release jitter (flat jitter profile). This is expected since as jitter disappears, all single condition tests converge to the same result, which is more pessimistic than that achieved with the n-conditions test. Test (3) seems usable for situations with low release jitter

and similar periods, only. In the general case, with a linear jitter profile, tests (2) and (4) seem to apply, presenting a tradeoff between pessimism, less with test (2), and computing overhead, less with test (4). As the release jitter becomes smaller and closer to the flat jitter profile, the performance of all tests improves substantially while their overheads stay approximately constant. Finally, it is also interesting to note the stronger impact that jitter has on RM scheduling with respect to EDF, which was also expected given the higher scheduling capacity of the latter.

5 Application Example

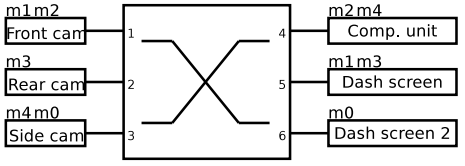
In order to illustrate the use of such fast schedulability tests in the context of dynamic QoS management, we will consider a case study developed around a fictitious automotive application for driver assistance relying on video transmissions [12]. To cope with the required bandwidth the system uses switched Ethernet [13] enhanced with the FTT-SE protocol, Flexible Time-Triggered communication over Switched Ethernet, to provide deterministic dynamic adaptation and reconfigurable virtual channels with guaranteed QoS. The purpose is to redistribute bandwidth released by cameras that are temporarily switched off among the remaining active cameras, improving the QoS that they get at each moment by maximizing the resource usage (links bandwidth).

In this application scenario several video sources are connected to the Ethernet switch, see Fig. 3a. The front cameras node produces two streams (m_1 and m_2) that are used for obstacle detection, lane departure warning and night vision, and which are connected to the dashboard screen and the video processing unit, respectively. The side cameras (streams m_0 and m_4) are used for obstacle detection in the blind spot, and are connected to the processing unit that may then trigger alarms, and to a secondary screen in the dash board, respectively. The rear view camera (stream m_3) connects to the dash board screen for driving assistance, when reversing.

The bandwidth distribution mechanisms follows a greedy approach that distributes the available bandwidth in a fixed priority order (according to parameter $P_{QoS,i}$). Each of the video streams operates under a constant frame rate of 25fps ($1/T_i$). The channel width ($C_i^w = C_i \times$ 100Mbps) can vary between 100kB (90kB for m_0) and 200kB. These values correspond to a bandwidth per channel that can vary between 20Mbps (18Mbps for m_0) and 40Mbps. Table 3b summarizes the streams properties.

The links have a capacity of 100Mbps each but the traffic scheduling model of FTT-SE imposes several constraints and overheads that reduce that capacity to 90Mbps. Nevertheless, it allows using any traffic scheduling policy, and considering full preemption. Therefore, we will use EDF, which allows us to make a full use of the 90Mbps of each link.

The streams generation model is periodic without jitter, which allows a direct use of the Liu and Layland utilization bounds [9] to assess the schedulability of the traffic in the uplinks (nodes to switch connection). Using EDF, the bound



(a) Architecture.

	$C_i \times 100Mbps$	T_i	$P_{QoS,i}$	S_i	R_i
m_0	90kB-200kB	40ms	0	3	6
m_1	100kB-200kB	40ms	1	1	5
m_2	100kB-200kB	40ms	2	1	4
m_3	100kB-200kB	40ms	3	2	5
m_4	100kB-200kB	40ms	4	3	4

(b) Requirements.

Fig. 3. Application scenario

equals the link capacity, i.e., 90Mbps, and since this is higher than the worst-case bandwidth requirement in any uplink, which is 2*40Mbps, the traffic is schedulable in any case.

However, as explained in [10], when there are several streams sharing an uplink but going to different downlinks, the traffic in the downlinks may arrive jittered. In fact, despite having similar periods, there is no synchronization between the cameras which send frames in self-triggered mode. This means that the frames of the streams can arrive at the switch with any (and variable) relative phase thus generating a variable interference pattern that leads to the referred jittered release in the downlinks. Consequently, the traffic schedulability in the downlinks must be assessed with the tests that account for release jitter as those discussed earlier in the paper.

In this case, we must start by determining the amount of jitter that may affect each stream. We can see that stream m_3 does not suffer interference on its uplink, arriving periodically at the switch, thus $J_3 = 0$. However, all other streams suffer interference of another stream in the respective uplink, thus creating an interference as large as one frame. Table 5 shows the release jitter that can affect each stream on downlinks $j = 4..6$. Note that when C_i changes so does J_i created by that stream.

Table 5.

	J_0	J_1	J_2	J_3	J_4
$j = 4$	x	x	C_1	x	C_0
$j = 5$	x	C_2	x	0	x
$j = 6$	C_4	x	x	x	x

A direct application of the schedulability tests (1) through (4), with the minimum requirements, will deliver a positive result in all downlinks. This means that the system can always meet those requirements. However, when testing schedulability with the maximum requirements all tests deliver a negative result in downlinks 4 and 5. This means it is not possible to satisfy the maximum requirements of all streams simultaneously and some adaptation must be put in place.

In this case, we use a simple greeady approach according to which we start reducing the QoS of the least priority streams until we can meet the resources capacity. At this point, a significant difference between tests (1) and (2) on one hand, and tests (3) and (4) on the other, is that with the former we do not know exactly the bandwidth in excess so that we directly determine the new distribution of bandwidth that allows us to meet the resource capacity. Basically, test (1) uses terms that do not directly represent bandwidth and test (2) uses n conditions and thus we cannot determine the bandwidth overload that must be removed from the system. Using these tests to carry out dynamic bandwidth management requires a trial and error approach that can be costly in number of iterations and thus implying high overhead. This, nevertheless, requires further assessment since it will substantially depend on the specific search technique used.

Therefore, we will use test (3), which, in a set with equal periods, is equivalent to test (4). In this case, the total utilization plus the virtual utilization corresponding to the jitter terms in both links 4 and 5 reaches 120Mbps, i.e., 30Mbps above the links capacity. In order to eliminate this overload we start reducing the QoS of m_0 to the minimum (18Mbps) which causes J_4 to be reduced accordingly (Table 5).

This has no impact on the overload and thus we need to reduce the QoS of m_1 to the minimum (20Mbps), leading to a corresponding reduction in J_2. This reduction from 40Mbps to 20Mbps impacts directly both links 4 and 5 and present now an overload of 10Mbps beyond the links capacities.

Thus, we still need to adjust the QoS of the following stream m_2 in order to reduce the 10Mbps overload from its QoS. Now, note that m_2 affects link 4 directly and link 5 indirectly vis J_2. Thus, by simply reducing the bandwidth of this stream from 40 to 30Mbps allows meeting exactly the link capacity of 90Mbps in both links 4 and 5. The final QoS assigned to each stream is given in Table 6.

Table 6.

	$P_{QoS,i}$	$U_{i,min}$	$U_{i,extra}$	U_i
m_4	4	$20Mbps$	$20Mbps$	$40Mbps$
m_3	3	$20Mbps$	$20Mbps$	$40Mbps$
m_2	2	$20Mbps$	$10Mbps$	$30Mbps$
m_1	1	$20Mbps$	$0Mbps$	$20Mbps$
m_0	1	$18Mbps$	$0Mbps$	$18Mbps$

Now, imagine that at a certain point in time both streams m_0 and m_1 are temporarily switched off. This implies that the associated jitter terms, J_0, J_1, J_2 and J_4, are nullified, which means the system becomes free of jitter. This is easy to see in Figure 3a since in such a configuration there will be one single stream coming from each node in each uplink. Running the schedulability test in all links will yield a positive result for the maximum requirements of the active

streams, which can thus operate at highest QoS, as shown in Table 7. Links 1, 2, 3 and 5 will be used at 40Mbps, link 4 will be used at 80Mbps and link 6 will be unused.

Table 7.

	$P_{QoS,i}$	$U_{i,min}$	$U_{i,extra}$	U_i
m_4	4	$20Mbps$	$20Mbps$	$40Mbps$
m_3	3	$20Mbps$	$20Mbps$	$40Mbps$
m_2	2	$20Mbps$	$20Mbps$	$40Mbps$

If one of the disconnected streams is connected again, the schedulability test must be executed for all links to determine whether that would create an overload and, if necessary, adjust the QoS of the active and arriving streams in order to make room for the arriving one while maximizing the usage of the links capacity and avoiding the overload.

6 Conclusion

Growing interest on improving the resource efficiency in embedded systems is pushing towards the use of on-line adaptation and reconfiguration. For example, dynamic bandwidth management, or more generally dynamic QoS management, may allow maximizing the use of a given resource, e.g. cpu or network, by a dynamic set of entities. When these systems have real-time requirements, the on-line adaptations must be carried out with the assistance of an appropriate schedulability analyzer that assures a continued timely behavior.

In this paper we have analyzed several utilization-based schedulability tests, one of which proposed herein, that incorporate release jitter. This feature was introduced very recently and allows their use in distributed systems where release jitter appears naturally. Moreover, they are particularly suited to be used on-line given their low computational overhead.

Therefore, we carried out an extensive simulation with random task sets to characterize the level of pessimism and computational overhead of the schedulability tests. We determined the most adequate utilization scenarios for each of the remaining tests. Finally, we illustrated the use of these tests in a fictitious automotive application involving the scheduling of several video streams.

Acknowledgments

This work was partially supported by the iLAND project, call 2008-1 of the EU ARTEMIS JU Programme and by the European Community through the ICT NoE 214373 ArtistDesign.

References

1. Zuhily, A., Burns, A.: Optimality of (D-J)-monotonic Priority Assignment. Information Processing Letters 103(6), 247–250 (2007)
2. Audsley, N., Burns, A., Richardson, M., Tindell, K., Wellings, A.J.: Applying New Scheduling Theory to Static Priority Pre-Emptive Scheduling. Software Engineering Journal 8, 284–292 (1993)
3. Bini, E., Baruak, S.K.: Efficient computation of response time bounds under fixed-priority scheduling. In: Proc. of 15th Int. Conf. on Real-Time and Networked Systems (RTNS'07) (March 2007)
4. Buttazzo, G.C.: Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications. Real-Time Systems Series. Springer, Santa Clara (2004)
5. Buttazzo, G.C., Lipari, G., Caccamo, M., Abeni, L.: Elastic Scheduling for Flexible Workload Management. IEEE Trans. on Computers 51(3), 289–302 (2002)
6. Davis, R.I., Burns, A.: Response Time Upper Bounds for Fixed Priority Real-Time Systems. In: Proc. of the 29th Real-Time Systems Symposium (RTSS'08), pp. 407–418. IEEE Computer Society, Washington (2008)
7. Fisher, N., Nguyen, T.H.C., Goossens, J., Richard, P.: Parametric Polynomial-Time Algorithms for Computing Response-Time Bounds for Static-Priority Tasks with Release Jitters. In: Proc. of the 13th IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA'07), pp. 377–385. IEEE Computer Society, Washington (2007)
8. Ghosh, S., Rajkumar, R.R., Hansen, J., Lehoczky, J.: Scalable QoS-Based Resource Allocation in Hierarchical Networked Environment. In: Proc. of the 11th IEEE Real Time on Embedded Technology and Applications Symposium (RTAS'05), pp. 256–267. IEEE Computer Society, Washington (2005)
9. Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the Association for Computing Machinery 20(1), 46–61 (1973)
10. Marau, R., Almeida, L., Pedreiras, P., Lakshmanan, K., Rajkumar, R.: Utilization-based Schedulability Analysis for Switched Ethernet aiming Dynamic QoS Management. In: Proc. of the 15th IEEE Int. Conf. on Emerging Technologies and Factory Automation ETFA'10 (September 2010)
11. Palencia, J.C., Harbour, M.G.: Response time analysis of EDF distributed real-time systems. J. Embedded Comput. 1(2), 225–237 (2005)
12. Rahmani, M., Steffen, R., Tappayuthpijarn, K., Steinbach, E., Giordano, G.: Performance analysis of different network topologies for in-vehicle audio and video communication. In: 4th International Telecommunication Networking Workshop on QoS in Multiservice IP Networks (IT-NEWS'08), pp. 179–184 (13-15, 2008)
13. Sommer, J., Gunreben, S., Feller, F., Kohn, M., Mifdaoui, A., Sass, D., Scharf, J.: Ethernet - A Survey on its Fields of Application. IEEE Communications Surveys Tutorials 12(2), 263–284 (2010)

Error Detection Rate of MC/DC for a Case Study from the Automotive Domain*

Susanne Kandl and Raimund Kirner

Institute of Computer Engineering
Vienna University of Technology, Austria
{susanne,raimund}@vmars.tuwien.ac.at

Abstract. Chilenski and Miller [1] claim that the error detection probability of a test set with full modified condition/decision coverage (MC/DC) on the system under test converges to 100% for an increasing number of test cases, but there are also examples where the error detection probability of an MC/DC adequate test set is indeed zero. In this work we analyze the effective error detection rate of a test set that achieves maximum possible MC/DC on the code for a case study from the automotive domain. First we generate the test cases automatically with a model checker. Then we mutate the original program to generate three different error scenarios: the first error scenario focuses on errors in the value domain, the second error scenario focuses on errors in the domain of the variable names and the third error scenario focuses on errors within the operators of the boolean expressions in the decisions of the case study. Applying the test set to these mutated program versions shows that all errors of the values are detected, but the error detection rate for mutated variable names or mutated operators is quite disappointing (for our case study 22% of the mutated variable names, resp. 8% of the mutated operators are not detected by the original MC/DC test set). With this work we show that testing a system with a test set that achieves maximum possible MC/DC on the code detects less errors than expected.

1 Introduction

Safety-critical systems are systems where a malfunction causes crucial damage to people or the environment. Examples are applications from the avionics domain or control systems for nuclear power plants. Nowadays also applications from the automotive domain become more and more safety-critical, for instance advanced driver assistance systems, like steer-by-wire or drive-by-wire. Safety-critical embedded systems have to be tested exhaustively to ensure that there are no errors in the system. The evaluation of the quality of the testing process is usually done with some code coverage metrics that determine the proportion of the program that has been executed within the testing process. For instance, a value of 60% for decision coverage means that 6 of 10 branches of all if-else decisions have been tested. Apart from very simple and small programs it is in general not possible to test all execution paths. This is especially true for programs with

* This work has been partially supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Sustaining Entire Code-Coverage on Code Optimization (SECCO)" under contract P20944-N13.

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 131–142, 2010.

complex boolean expressions within the decisions. For a decision depending on a condition that consists of n boolean subconditions we would need to generate 2^n inputs to test all possible combinations. That means the testing effort would grow *exponentially* with increasing complexity of the condition within the decision.

Modified condition/decision coverage (MC/DC) is a metric originally defined in the standard DO-178B [2], a standard for safety-critical systems in the avionics domain. In principle MC/DC defines that the set of test data has to show that each condition within a decision can independently, i.e., while the outcome of all other conditions in the decision remain constant, influence the outcome of the decision. It can be shown that MC/DC needs only $n+1$ test cases to test a decision that contains n conditions. Thus the testing effort grows only *linearly* with the number of conditions per decision. Until now MC/DC was mainly an important coverage metric for applications from the avionics domain. Due to the fact that more and more applications for cars are also high-safety critical and because of the fact that these applications become even more complex, the coverage metric MC/DC is now also an issue for the testing process for components from the automotive domain. Existing standards for safety-critical systems for cars are IEC 61508 [3] or ISO 26262 (Road vehicles - Functional safety) [4] which is available as a draft, the final version is expected for 2011.

An ideal testing process is capable of finding any error within the system. Our aim in practice is a test set consisting of the smallest possible number of test cases that is able to detect as many errors as possible. MC/DC is seen as a suitable metric to evaluate the testing process of safety-critical systems with a manageable number of test cases. But what about the effective error detection rate of MC/DC for a real case study? In [1] it is stated that the error detection probability of MC/DC is nearly 100% for an increasing number of test cases. On the other hand side there are examples containing coding errors for which the probability of detecting an error with an MC/DC adequate test set is actually zero, see [5]. This contradiction motivated us to evaluate the error detection rate of an MC/DC adequate test set for a real case study from the automotive domain. Our goal was to find out how the coverage correlates with the error detection rate and to prove whether a test set with full MC/DC on the code is capable to find the errors.

For our experiments we define three error scenarios: In the first scenario only concrete *values* for output variables are changed. The second error scenario focuses on errors in the *names* of output variables. The third error scenario considers errors of the *operators* in the boolean expressions in the decisions. The test cases are generated automatically with a model checker to confirm a suitable MC/DC test set. The test runs are executed with the mutated program versions and the MC/DC adequate test set. The results show that a MC/DC adequate test set is capable to reveal all errors of concrete values, but it fails in detecting errors for variable names or operators.

The paper is organized as follows: In the following section we recapitulate the coverage metric MC/DC. Subsequently we describe our test case generation method. In Section 5 we describe in detail how the program versions for our test runs are mutated. Section 6 shows our experimental results. In the concluding section we discuss what these results mean for practice, i.e. how applicable MC/DC is for the evaluation of the testing process for safety-critical systems in the automotive domain.

2 Unique-Cause MC/DC

MC/DC is a code coverage metric introduced in DO-178B [2], discussed in detail in [6], resp. expanded with variations of the metric in [7]. The metric is a structural coverage metric defined on the source code and is designed to test programs with decisions that depend on one or more conditions, like `if` $((A \wedge (B \vee C))$ `statement_1 else statement_2`.

In MC/DC we need a set of test cases to show that changing the value for each particular condition changes the outcome of the total decision independently from the values of the other conditions. (This works as long there is no coupling between different instances of conditions.)

A test suite conforming to MC/DC consists of test cases that guarantee that [2], [6]:

- every point of entry and exit in the model has been invoked at least once
- every basic condition in a decision in the model has been taken on all possible outcomes at least once, and
- each basic condition has been shown to *independently* affect the decision's outcome.

The independence of each condition has to be shown. If a variable occurs several times within a formula each instance of this variable has to be treated separately, e.g. for $(A \wedge B) \vee (A \wedge C)$ beside the independence of B and C the independence of A has to be shown for the first occurrence and the second occurrence of A. Independence is defined via *Independence Pairs*. For details please refer to [7].

Consider the example $A \wedge (B \vee C)$: The truth table is given in Table 1 (third column). In the following $\bar{0}$ represents the test case $(0, 0, 0)$, $\bar{1}$ represents the test case $(0, 0, 1)$, and so on. The independence pairs for the variable A are $(\bar{1}, \bar{5})$, $(\bar{2}, \bar{6})$ and $(\bar{3}, \bar{7})$, the independence pair for the variable B is $(\bar{4}, \bar{6})$ and the independence pair for the variable C is $(\bar{4}, \bar{5})$. Thus we have the test set for MC/DC consisting of $\{\bar{4}, \bar{5}, \bar{6}\}$ plus one test case of $\{\bar{1}, \bar{2}\}$ (remember that for n conditions we need $n + 1$ test cases).

Table 1. Truth Table for Different Boolean Expressions

Testcase	*A B C*	$A \wedge (B \vee C)$	$(A \wedge B) \vee C$	$(A \wedge B) \oplus C$
$\bar{0}$	0 0 0	0	0	0
$\bar{1}$	0 0 1	0	1	1
$\bar{2}$	0 1 0	0	0	0
$\bar{3}$	0 1 1	0	1	1
$\bar{4}$	1 0 0	0	0	0
$\bar{5}$	1 0 1	1	1	1
$\bar{6}$	1 1 0	1	1	1
$\bar{7}$	1 1 1	1	1	**0**

3 Error Detection Probability - Theoretical Assumption and Counterexample

In [1] different code coverage metrics are compared and a subsumption hierarchy for the most relevant code coverage metrics is given. It is stated that "the modified condition/decision coverage criterion is more sensitive to errors in the encoding or compilation of a single operand than decision or condition/decision coverage", as we can see in Figure 1.

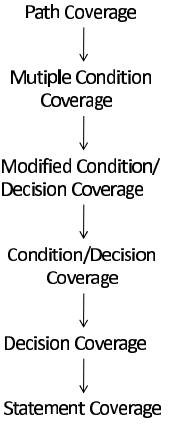


Fig. 1. Subsumption Hierarchy for Control Flow Metrics [1]

Moreover the probability of detecting an error is given as a function of tests executed. For a given set of M distinct tests, the probability $P_{(N,M)}$ of detecting an error in an incorrect implementation of a boolean expression with N conditions is given by [1]

$$P_{(N,M)} = 1 - \left[\frac{2^{(2^N - M)} - 1}{2^{2^N}} \right].$$

This correlation is shown in Figure 2 for $N = 4$.

One important result of this relation is the relatively low probability of detecting errors with only two test cases, as normally required in decision or condition decision testing. As M increases there is a rapid increase in the error detection probability. As N grows, $P_{(N,M)}$ rapidly converges to $1 - 1/2^M$ and the sensitivity changes only marginally with N. That means for N increasing the likelihood of detecting an error in an expression of N conditions with $N + 1$ test cases increases also. This non-intuitive result occurs because the dominant factor (the number of tests) increases with N while the sensitivity to errors remains relatively stable.[1]

3.1 Counterexample for Theoretical Assumption

In contrast to the assumption above one can easily construct examples similar to [5] that show that the error detection probability is actually zero for a given decision with

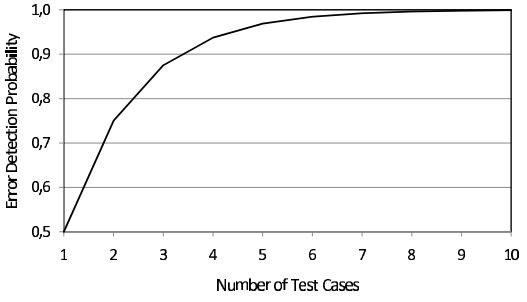


Fig. 2. Error Detection Probability of MC/DC [1]

a complex boolean expression. The example shows that for $(A \wedge B) \vee C$ the test set for achieving full MC/DC consists of the test cases $\{\bar{2}, \bar{4}, \bar{6}\}$ plus one test case out of $\{\bar{1}, \bar{3}, \bar{5}\}$. Mutating the logical operator *OR* to a logical *XOR* changes the output only for the test case $\bar{7}$, i.e. for the combination True, True, True for the variables A, B and C, thus the occurring error is not detected with a given minimal MC/DC test set. (See also the last two columns in the Table 1.) Furthermore it can also be shown that a test set for decision coverage would detect the error with a probability of 20%, so this is also a counterexample for the statement that MC/DC subsumes decision coverage.

We were interested if such counterexamples are only artificial outliers that have no influence on the overall error detection probability of MC/DC test sets or if such counterexamples really decrease the error detection probability of MC/DC. First we constructed a few examples manually for small programs that showed that in the worst case even about 30% of operator errors were not detected with a MC/DC adequate test set. This result was quite alarming for testing safety-critical systems. Then we executed our test runs on a real case study from the automotive domain. The results are given in Section 6. In the following we describe our test case generation method and the error scenarios.

4 Test Case Generation

The generation of test cases that result in full MC/DC is a non-trivial issue. As we have seen in Section 2 the test cases have to be determined by deriving the independence pairs for each sub-condition. We generate the test cases automatically with a model checker [8], in our case NuSMV[1].

4.1 Principle

A model checker takes a model of the system under test and proves whether a given property is valid within the model or not. In the case of a violation of a given property

[1] http://nusmv.irst.itc.it

the model checkers produces a *counterexample*, that means a trace where the property is violated. This trace is a concrete execution path for the program we want to test and can therefore be used as a test case. The big challenge for generating suitable test cases consists mainly in the method *how* the trap properties (the properties that produce counterexamples that can be used as test cases) are formulated.

4.2 Method

Our test case generation method is motivated by works from Whalen et al [9]. In this paper a metric called *Unique First Cause Coverage (UFC)* is introduced. This metric is adapted from the MC/DC criterion and is defined on LTL (linear temporal logic) formulas. For a given set of requirements for a program to test we have to translate the requirements into a formal language (in our case LTL). Then the trap properties for deriving an MC/DC adequate test set are directly derived from these formal properties by mutation with rules that are similar to the definition of MC/DC on the code. That means if we have a sufficient set of trap properties, the model checker produces a complete test set for full MC/DC.

4.3 Example

We want to demonstrate the test case generation method on the following example. Listing 1.1 shows a small C program, for which the corresponding NuSMV-model is given in Listing 1.2. In the program we have a decision which evaluates to True (`res=42`) or False (`res=24`) depending on the boolean expression $A \wedge (B \vee C)$ similar to the example in Section 2. The NuSMV-model represents the behavior of the program in the automaton language of NuSMV. After the declaration of the variables within the ASSIGN block the model calculates `res` depending on the validity of the boolean expression. The specification for this small program consists only of two requirements, from these we can derive the properties we need to generate the test cases (MC/DC-trap properties in Listing 1.2). With the given trap properties we gain the test set for MC/DC consisting of $\{\bar{2}, \bar{4}, \bar{5}, \bar{6}\}$. Applying these test cases to the original implementation results in full MC/DC.

4.4 Unreachable Code

Applying this test case generation method to our case study from the automotive domain showed that some of the formulated trap properties are true within the model, that means that no counterexample is produced. Analyzing these cases showed that there are a few infeasible paths in the program caused by a program structure depicted in Listing 1.3. In this program structure there is no possible combination for the input variables a and b to reach the else-branch of the decision in line 7. (with 0,0 statement_2 is executed, with 0,1 also statement_2, with 1,0 - statement_3 and with 1,1 statement_1 is executed).

```
1 #include <stdio.h>
2 typedef int bool;
3 int erg;

5 int test(bool a, bool b, bool c)
6 {
7   if (a && (b || c))
8     res = 42;
9   else
10    res = 24;
11 }

13 int main()
14 {
15   test(0,0,1);
16   printf("Result: %d \n", res);
17 }
```

Listing 1.1. C Source Code

```
1 MODULE main
2 VAR  -- Variables
3  a: boolean;
4  b: boolean;
5  c: boolean;
6  res: {0, 42, 24};
7 ASSIGN
8   init(res) := 0;
9   next(res) := case
10    a & (b | c): 42;
11    !(a & (b | c)): 24;
12    1: res;
13  esac;

15 -- REQUIREMENTS - original
16 PSLSPEC AG(a&(b|c)->AX(res=42));
17 PSLSPEC AG(!(a&(b|c))->AX(res=24));

19 -- MC/DC - trap properties
20 PSLSPEC AG(a&(!b|!c)->AX!(res=24));
21 PSLSPEC AG(a&(!b|c)->AX!(res=42));
22 PSLSPEC AG(a&(b|!c)->AX!(res=42));
23 PSLSPEC AG(!a&(b|c)->AX!(res=24));
```

Listing 1.2. NuSMV Model

4.5 Test Traces vs. Test Steps

The described test case generation method produces complete traces within the program. A test *trace* consists of multiple test *steps*. A test step is a mapping between input data and the expected output for a decision. See the example given in Listing 1.3. If we want to test the decision in line 4 we need a trace to statement_2 (line 5) and a trace to the else-branch of this decision (line 6). For testing the if-decision in line 7 again we need a trace to the else-branch of the previous decision (line 6) to reach the if-decision in line 7 and to execute the corresponding statement_3 (line 8). This yields the side effect that the generated test set is redundant in that way that a) the same statement (e.g., statement_2) maybe executed several times and b) also the corresponding values are checked multiple. For a minimal MC/DC test set we only need the test data for the different decisions, so we can reduce the test traces to singular test steps. So we reduced the generated traces to the necessary test steps to gain a *minimal* MC/DC test set (without redundant test data).

For our testing process it is important to mention that we test not only the input-output mappings of the program, but also the values of the internal variables. Consider the following case: For given input data the output in the testing process conforms to the expected output, but nevertheless there is an erroneous value during the calculation of the output values. This case would be also detected within the testing process. The

evaluation of the coverage was done with Tessy[2]. In the overall with the generated test set we achieve the maximal possible MC/DC coverage of 92,77%.

```
1 if (a && b)
2   statement_1
3 else
4     if (!a)
5       statement_2
6     else
7       if (!b)
8         statement_3
```

Listing 1.3. Unreachable Code

5 Case Study and Error Scenarios

For our experiment we define three different error scenarios:

- **Value Domain:** The first error scenario investigates how many errors are detected by the produced test set for a erroneous value, i.e. for a given variable `variable_1` with a specified value of 2, we change the specified value to the value, for instance, 3.
- **Variable Domain:** The second error scenario checks how many errors are detected by the produced test set if there are erroneous variable names for the output variables within the implementation, for instance, in the program there exist 2 variables with the names `variable_1` and `variable_2` we change some occurrences of the variable `variable_1` to the name of the other variable `variable_2` and vice versa. (In that case we just have to take care that these changes are compatible with the referring data types of the different variables.)
- **Operator Domain:** The third error scenario focuses on coding errors for the operators of the boolean expressions within the decisions. The given test set is executed on the erroneous program versions to find out if the test cases succeed or fail. For the example from Listing 1.1 we may change the first operator in $A \wedge (B \vee C)$ from the logical *AND* to the logical *OR* and vice versa for the second occurring operator.

We use the term *error* for some program property that differs from the correct implementation due to the system specification. Our case study is a low safety-critical control system from the automotive domain. It regulates a steering mechanism controlled by various input values through multiple output values, these output values are dependent on the input values and the program's internal values. To give a draft of the complexity of the program the control flow graph is given in Figure 3.

For the test runs the original program is mutated systematically on basis of the predefined error scenarios and the resulting program versions were executed with the given MC/DC test set. If at least one test case fails in the test run, we know that the coding error has been detected, otherwise if all test cases run successfully we register that the error has not been detected in the test run.

[2] http://www.hitex.com/index.php?id=module-unit-test

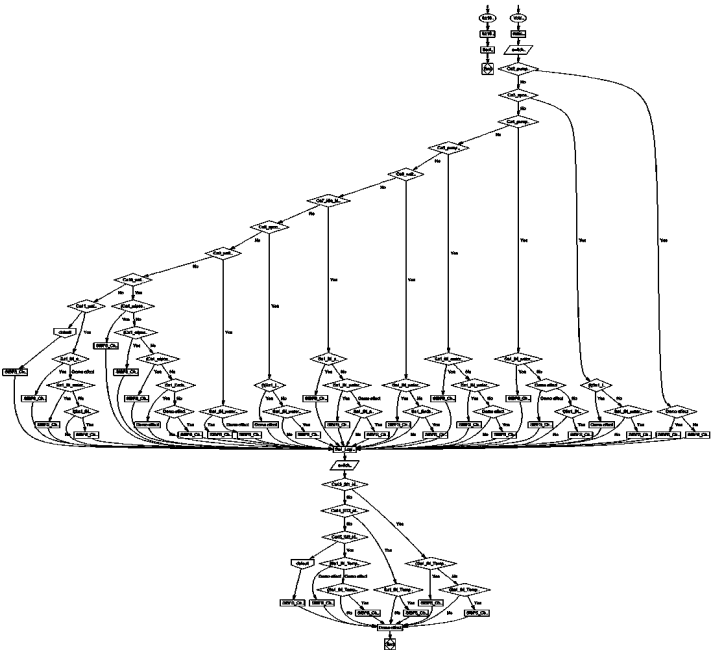


Fig. 3. Case Study - Control Flow Graph

6 Experimental Results of the Testing Process

Remember that with our test case generation method we gained a minimal test set that achieves 92,77% MC/DC on the code. This is the maximum value for the achievable coverage due to some parts of unreachable code. We mutated the original program referring to the three error scenarios (Value Domain *Err_Val*, Variable Domain *Err_Var* and Operator Domain *Err_Op*) and executed it with the test sets for different values of MC/DC, namely a test set with 20%, 40%, 60%, 80% and the maximum of 92,77% coverage. The following table shows the percentage of detected errors, i.e. a value of 28% means that 28 out of 100 error were detected. The results are also given in the diagram Figure 4.

7 Discussion of the Results

As we can see the error detection rate for errors in the value domain increases like expected with the increasing coverage, resp. with a larger test set. For the maximum possible coverage the error detection rate is indeed 100% for our case study. This means full MC/DC coverage guarantees that an error will be detected with a given test set.

Table 2. Error Detection Rate (in %) for Test Sets with Different MC/DC Coverage

Coverage	*Err_Val*	*Err_Var*	*Err_Op*
20	16	14	40
40	36	28	58
60	46	40	80
80	70	56	90
max	100	78	92

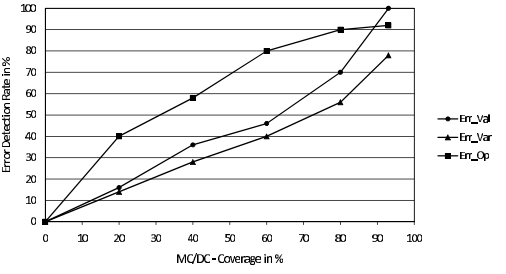


Fig. 4. Experimental Results

The results for errors in the variable and the operator domain differ significantly from that. The error detection rate for errors in the variable domain also increases with the coverage, resp. with a bigger set of test cases, but with the complete test set there are still 22% of errors undetected. Looking at the original program we see that there are several statement blocks where different variables are assigned. See Listing 1.4. If the name of the variable_1 in line 2 is changed to the name variable_2, the assignment of value_1 to the variable_1 gets lost, that means the value for variable_2 is still correct, the value for variable_1 may be correct (depending on the initial value before the assignment in line 2). Vice versa if we mutate the name of variable_2 in line 3 to variable_1, the value of variable_1 is overwritten with value_2. This may also be undetected for the case value_1 equals value_2, which is an improbable coincidence for integer values but quite thinkable for boolean values. This may be an explanation for such a high amount of undetected errors for variable names.

```
1 if (a && b)
2     variable_1 = value_1
3     variable_2 = value_2
```

Listing 1.4. Mutation Variable Name

For the third error scenario with mutated operators for the boolean expressions within decisions we see that already a test set with low coverage is capable to identify many errors, for instance the test set with 40% coverage already finds more than the half of errors. But still with a complete MC/DC test 8% of erroneous operators remain undetected. This value demonstrates the empirical evaluation of the example given in Section 3.1. We think that 22, resp. 8 of 100 undetected errors for a safety-critical system is quite risky.

8 Related Work

Besides the original definition of the code coverage metric MC/DC in the standard DO-178B [2] and the corresponding document *Final Clarification of DO-178B* [6] the

metric is discussed and extended in [7]. The applicability is studied in Chilensky and Miller [1], in this document an assumption for the error detection probability is given but not proved with empirical data. An empirical evaluation of the MC/DC criterion can be found in [10]. Although the most important issue for the quality of a code coverage metric for safety-critical systems is indeed the capability of detecting an error, it is surprising that there are hardly any empirical studies of the error detection probability of MC/DC for real case studies. A principal comparison of structural testing strategies is [11], whereas an empirical evaluation of the effectiveness of different code coverage metrics can be found in [12]. Also in [13] MC/DC is compared to other coverage criteria for logical decisions. Rajan et al. show in [14] that even the structure of the program (for instance, if a boolean expression is inlined or not) has an effect on the MC/DC coverage. The introduced test case generation method using model checkers is described in Gargantini et al. [15], Hamon et al. [16] and Okun et al. [17] and [18]. For the test case generation of MC/DC adequate tests the trap properties have to be formulated to enforce the model checker to produce the appropriate paths we need to achieve MC/DC. This is discussed in Raydurgan and Heimdahl [19] or Whalen et al. [9]. The last work gives results that show that for a given complete test set for MC/DC (derived from the implementation or model of the SUT) does not achieve full MC/DC caused by special structure in the code, for instance, macros, that where not considered in the model used for test case generation.

9 Summary and Conclusion

A testing process is only as good as it is capable to reveal errors within the system under test. Coverage metrics like MC/DC are a means of evaluating the testing process, i.e. to measure which parts of the program have been executed within the testing process. In the standard DO-178B a high-safety critical system has to be tested with a test set that achieves full MC/DC coverage on the code. Recently upcoming standards like ISO 26262 will also prescribe this metric for safety-critical applications from the automotive domain. In this work we have shown that by achieving full MC/DC during testing it is not guaranteed that the probability of undetected errors is sufficiently low concerning reliability requirements for safety-critical system. An MC/DC adequate test set seems to be capable to reveal all errors in the value domain, but many errors concerning erroneous variable names or erroneous operators are not detected with this test set: for our case study 22%, resp. 8% of the errors were not detected which is really precarious for a safety-critical system. Similar works (e.g., [14]) also show that MC/DC is not robust to structural changes in the implementation.

Overall it is important to be aware of that so far MC/DC is the best metric for testing safety-critical system with complex boolean expressions within decisions referring to the tradeoff between testing effort (number of test cases to achieve full coverage) and efficiency in the error detection rate, *but* although achieving full MC/DC coverage there may be still a high amount of errors undetected.

References

1. Chilenski, J., Miller, S.: Applicability of modified condition/decision coverage to software testing. Software Engineering Journal 9(5), 193–200 (1994)
2. RTCA Inc.: DO-178B: Software Considerations in Airborne Systems and Equipment Certification. Requirements and Technical Concepts for Aviation, Washington, DC (December 1992)
3. International Electrotechnical Commission: IEC 61508: Functional Safety of Electrical Electronic Programmable Safety-Related Systems (1999)
4. ISO: International Organization for Standardization: ISO 26262: Functional safety – road vehicles, draft (2009)
5. Bhansali, P.V.: The MCDC paradoxon. SIGSOFT Softw. Eng. Notes 32(3), 1–4 (2007)
6. RTCA Inc.: DO-248B: Final Report for Clarification of DO-178B: Software Considerations in Airborne Systems and Equipment Certification. Requirements and Technical Concepts for Aviation, Washington, DC (October 2001)
7. Chilenski, J.J.: An investigation of three forms of the modified condition decision coverage (MCDC) criterion. U.S. Department of Transportation, Federal Aviation Administration, DOT/FAA/AR-01/18 (April 2001)
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (2000)
9. Whalen, M.W., Rajan, A., Heimdahl, M.P., Miller, S.P.: Coverage metrics for requirements-based testing. In: ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis, pp. 25–36. ACM, New York (2006)
10. Dupuy, A., Leveson, A.: An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In: Digital Aviation Systems Conference (October 2000)
11. Ntafos, S.: A comparison of some structural testing strategies. IEEE Transactions on Software Engineering 14(6), 868–874 (1988)
12. Kapoor, K., Bowen, J.: Experimental evaluation of the variation in effectiveness for DC, FPC and MC/DC test criteria. In: 2003 International Symposium on Empirical Software Engineering, ISESE 2003. Proceedings, September 1-October, pp. 185–194 (2003)
13. Yu, Y.T., Laub, M.L.: A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. Journal of Systems and Software 79(5), 577–590 (2006)
14. Rajan, A., Whalen, M.W., Heimdahl, M.P.: The effect of program and model structure on MC/DC test adequacy coverage. In: ICSE '08: Proceedings of the 30th International Conference on Software Engineering, pp. 161–170. ACM, New York (2008)
15. Gargantini, A., Heitmeyer, C.: Using Model Checking to Generate Tests From Requirements Specifications. In: 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 146–162 (1999)
16. Hamon, G., de Moura, L., Rushby, J.: Generating Efficient Test Sets with a Model Checker. In: Proceedings of the 2nd International Conference on Software Engineering and Formal Methods, pp. 261–270 (2004)
17. Okun, V., Black, P., Yesha, Y.: Testing with model checkers: Insuring fault visibility (2003)
18. Okun, V., Black, P.E.: Issues in software testing with model checkers (2003)
19. Rayadurgam, S., Heimdahl, M.P.: Generating MC/DC adequate test sequences through model checking. In: Proceedings of the 28th Annual IEEE/NASA Software Engineering Workshop – SEW-03, Greenbelt, Maryland (December 2003)

Simultaneous Logging and Replay for Recording Evidences of System Failures

Shuichi Oikawa and Jin Kawasaki

Department of Computer Science, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki 305-8573, Japan

Abstract. As embedded systems take more important roles at many places, it is more important for them to be able to show the evidences of system failures. Providing such evidences makes it easier to investigate the root causes of the failures and to prove the responsible parties. This paper proposes simultaneous logging and replaying of a system that enables recording evidences of system failures. The proposed system employs two virtual machines, one for the primary execution and the other for the backup execution. The backup virtual machine maintains the past state of the primary virtual machine along with the log to make the backup the same state as the primary. When a system failure occurs on the primary virtual machine, the VMM saves the backup state and the log. The saved backup state and the log can be used as an evidence. By replaying the backup virtual machine from the saved state following the saved log, the execution path to the failure can be completely analyzed. We developed such a logging and replaying feature in a VMM. It can log and replay the execution of the Linux operating system. The experiment results show the overhead of the primary execution is only fractional.

1 Introduction

Commodity embedded systems, such as mobile phones, car navigation systems, HD TV systems, and so on, became so complicated that it is almost impossible to make them free from defects. Since there are a large number of usage patters, manufactures are not able to perform tests that cover all possible usage patters in advance. Since users tend to operate those commodity embedded systems for a long time, users cannot remember exactly what they did when system failures occurred. In such situations, it is not clear which side is responsible for the system failures, and both sides can blame each other for the causes of the failures.

As those embedded systems take more important roles at many places for many people, there are more chances that their failures may directly connect to financial loss or physical damage. System failures do not mean only software failures, but they mean the whole other failures of the parts included in systems. When a software malfunction happens, it may affect mechanical parts and can damage something or somebody. System failures may not be considered to be failures at first. Because of the perceptions by customers and/or societies, they can suddenly become failures. Therefore, it is important to record and provide the evidences of system failures. The evidences make it easier to investigate the root causes of the failures and to prove the responsible parties.

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 143–154, 2010.

We propose a system that enables the complete tracing of system failures and such execution traces are used as their evidences. We employ two virtual machines, one for the primary execution and the other for the backup execution. Those two virtual machines run on a virtual machine monitor (VMM) [5,9] of a single system. The VMM records the primary execution, and generates an execution log in a buffer. Then, on the backup virtual machine, the VMM replay the primary execution following the log in the buffer. The backup execution is exactly the same as the primary execution as far as the executed instruction stream and the produced values are concerned. Maintaining the log at a certain size creates a time lag for the backup execution. In other words, the execution of the backup virtual machine follows the execution of the primary virtual machine, but the backup execution never catches up the primary execution by buffering the execution log. Thus, the backup virtual machine maintains the past state of the primary virtual machine along with the log to make the backup the same state as the primary. When a system failure occurs on the primary virtual machine, the VMM saves the backup state and the log. By replaying the backup virtual machine from the saved state following the saved log, the execution path to the failure can be completely analyzed and provides the concrete evidence of the system failure.

We developed such a logging and replaying feature in a VMM. The VMM is developed from scratch to run on an SMP PC compatible system. It can log and replay the execution of the Linux operating system. The experiments show that the overhead of the primary execution is only fractional, and the overhead of the replaying execution to boot up the backup is less than 2%. This paper presents the detailed design and implementation of the logging and replaying mechanisms.

The rest of this paper is organized as follows. Section 2 shows the overview of the propsed system architecture. Section 3 describes the rationale of the logging and replaying of the operating system execution. Section 4 describes the design and implementation of the logging and replaying mechanisms. Section 5 describes the current status and the experiment results, and Section 6 discusses the current issues and possible improvements. Section 7 describes the related work. Finally, Section 8 concludes the paper.

2 System Overview

This section describes the overview of the propsed system architecture. Figure 1 shows the overview of the architecture. The system runs on an SMP system with 2 processors. In the figure, there are 2 processor cores, Core 0 and 1, which share physical memory. The primary VMM runs on Core 0, and creates the primary virtual machine. The primary virtual machine executes the primary guest OS. Core 1 is used for the backup. The primary and backup VMMs are the same except that they run on different cores. The memory is divided into three parts, one for the primary, another for the backup, and the last one for the shared memory between the primary and the backup.

Users use the the primary guest OS. It interacts with devices through the primary VMM; thus, it is executed just as an ordinary guest OS that runs on a VMM. Only difference is that the primary VMM records the events described in the previous section, so that its execution can be replayed on the backup.

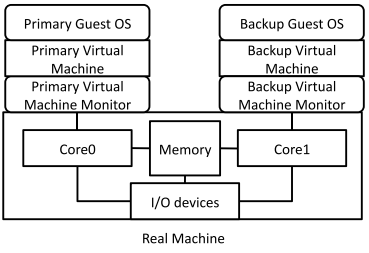


Fig. 1. Overall Architecture of the Proposed System

The execution log is transferred from the primary to the backup via shared memory. For every event that has to be logged, the primary VMM writes its record on the shared memory, and the backup VMM reads the record. The mapping is created at the boot time of the VMM. The shared memory is used only by the VMM since the log data is written and read by the VMM not by the Linux; thus, the existence of the shared memory region is not notified to the Linux. The size of the shared memory is currently set to 4 MB. The size should be adjusted to an appropriate value considering system usage.

The backup VMM reads the execution log from the shared memory, and provides the backup guest OS with the same events for the execution for the replaying. The backup guest OS does not take any inputs from devices but takes the replayed data from the backup VMM. The outputs produced by the backup guest OS are processed by the backup VMM. They can be output to a different unit of the same device or to an emulated device.

3 Rationale

This section describes the rationale to realize the logging and replaying of the operating system execution, and clarifies what needs to be logged. The basic idea of the logging and replaying of instruction execution is about the treatment of the factors outside of programs. Those factors include inputs to programs and interrupts, and both of them are events external to programs. This paper describes the rationale specific to the IA-32 architecture [6].

If a function contains the necessary data for its computation and does not interact with the outside of it, it always returns the same result regardless of the timing and the state of its execution. If a function accesses an I/O port and reads a value from it, the result can be different depending upon the value read from the I/O port. Therefore, in order to make the instruction execution streams and the produced values the same on the primary and the backup, the values copied from I/O ports must be recorded on the primary, and the same value must be provided from the corresponding port in the same order on the backup.

External interrupts change instruction execution streams, and external interrupts are caused by the factors that are outside of the executed program. Furthermore, external interrupts are vectored requests. There are interrupt request numbers (IRQ#) that are

associated with devices. Different IRQ# can be used in order to ease the differentiation of interrupt sources. Therefore, in order to make the instruction execution streams the same on the primary and the backup, the specific points where external interrupts are taken in the instruction execution stream and their IRQ# must be recorded on the primary, and the interrupts with the same IRQ# must be caused and taken at the same points on the backup.

The problem here is a way to specify a point where an external interrupt is taken. If there is no branch instruction in a program, an instruction address can be used to specify the point. There are, however, branches in most programs; thus, the number of branches needs to be counted. The IA-32 architecture includes instructions that loops within a single instruction, and does not change the number of branches. REP instruction repeats some types of a load or store instruction for the number of times specified in %ecx register. The repeating operation can be suspended by an interrupt, and be resumed again after the interrupt processing is finished. Therefore, the value of %ecx register also needs to be recorded. Therefore, by recording the instruction address, the number of branches since the last interrupt, and the value of %ecx register, it is possible to specify the point where the interrupt is replayed.

4 Logging and Replay

This section describes the design and implementation of the logging and replaying mechanism on the proposed system architecture. Our VMM is implemented on the IA-32 processor with Intel VT-x [8] feature. We take advantage of the capabilities available only to Intel VT-x for the efficient handling of the logging and replaying.

4.1 Logging

We first describe the retrieval of the information needed for the logging, and then its recording. There are the two types of the events that need to be logged, IN instructions and external interrupts. Event types can be captured from the VM exit reason. Intel VT-x implements the hardware mechanism that notifies the VMM about guest OS events. A VM exit is a transfer of control from the guest OS to the VMM, and it comes with the reason. By examining the VM exit reason, the VMM can distinguish which type of an event happened.

For an IN instruction event, the VMM needs to capture the result of the instruction execution. In order to do so, the VMM obtains the control when the guest OS executes an IN instruction, and executes the IN instruction with the same operand on behalf of the guest OS. Intel VT-x has the setting that causes a VM exit when an IN instruction is executed in the guest OS. The primary processor-based VM-execution controls define various reasons that cause a VM exit. Bit 24 of the controls determines whether the executions of IN and OUT instructions cause VM exits. By setting that bit, the VMM can obtain the control when the guest OS executes an IN instruction. The I/O port number that the IN instruction is about to access is provided in the VM exit qualification. The result of an IN instruction execution is returned in %eax register. The VMM copies the value to the guest %eax register, and also records the value in the log along with the event type.

```
typedef struct {
    unsigned long index;
    unsigned long address;
    unsigned long ecx;
    unsigned long reserved;
    unsigned long result_h;
    unsigned long result_l;
    unsigned long counter_h;
    unsigned long counter_l;
} RingBuffer;
```

Fig. 2. Definition of the Ring Buffer Entry

For an external interrupt event, the VMM needs to capture the IRQ#, the instruction address where the interrupt happened, the number of branches since the last event, and the value of %ecx register. The IRQ# is available from the VM-Exit interruption-information field. The instruction address of the guest OS is defined in the guest register state. The number of branches is counted by the Performance Monitoring Counter (PMC). The IA-32 architecture has the PMC facilities in order to count a variety of event types for performance measurement. Those event types include branches. The value of %ecx register is saved by the VMM at a VM-exit.

The retrieved event record is saved in an entry of the ring buffer constructed on the shared memory. Figure 2 shows the definition of the ring buffer entry structure in the C programming language. The fixed size structure is used for the both types of the events. Structure members `index`, `addess`, and `ecx` contain the type of an event, the instruction address where the event happened, and the value of %ecx register, respectively. The result value associated with an event can hold up to 8 bytes (64 bits) in `result_h` and `result_l`. `counter_h` and `counter_l` contain the number of branches since the last event. A PMC register can be up to 8 bytes (64 bits) long, and it is 40 bits long for the processor we currently use for the implementation; thus, 2 unsigned long members are used to contain the number of branches. The size of the structure is 32 byte, which is well aligned with the IA-32 processor's cache line size of 64 byte.

The ring buffer on the shared memory is managed by 3 variables, `ringbuf`, `write_p`, and `read_p`. Variable `ringbuf` points to the virtual address of the shared memory. The 4 MB region for the shared memory is statically allocated by the configuration; thus, the variable is initialized to the fixed value at the boot time. Variable `write_p` points to the index variable that indicates where a new event is recorded. Variable `read_p` points to the index variable that indicates where an unread event is stored. Because those indices are shared by the logging side and the replaying side and they need to be shared, `write_p` and `read_p` and pointer vaiables.

4.2 Replaying

We describe the execution replaying following the recorded log. In order to replay an event, the VMM first reads the next event record from the shared memory. The read event record contains the event type information. The event type determines the next

action for the VMM to do. As described above, there are two types of the events, an IN instruction event and an external interrupt event. We first describe the replaying of an IN instruction event, and then an external interrupt event.

In order to replay an IN instruction event, the VMM needs to obtain the control when the backup guest OS executes an IN instruction. The VMM on the backup sets up Intel VT-x in the same way as the primary to cause a VM exit when an IN instruction is executed in the guest OS. The record of an IN instruction event contains the data returned by the IN instruction on the primary. The VMM does not execute an IN instruction on behalf of the guest OS. The VMM simply returns the recorded data, instead, as the data read by the emulated IN instruction. We can make the order of the IN instructions execution the same on the backup as the primary as long as the results returned by the IN instructions are maintained the same.

The replaying of an external interrupt event is much harder. It is because an external interrupt can happen everywhere an interrupt is not disabled. In other words, there is no specific instruction that specifies where an external interrupt happens. Therefore, the VMM relies upon the information recorded on the primary and needs to point out the place and timing where an external interrupt event is injected into the guest OS.

The VMM determines the point to inject an external interrupt event by the following steps:

1. Execute the guest OS until the instruction address where the interrupt is supposed to be injected.
2. Compare the numbers of branches of the guest OS and the event record.
   (a) Proceed to the next step if they matches. Go back to Step 1 above if they don't.
3. Compare the values of %ecx register of the guest OS and the event record.
   (a) Proceed to the next step if they matches. Execute a single instruction and go back to Step 3 if they don't.
4. Inject the interrupt into the guest OS.

At Step 1, the VMM needs to obtain the control when the guest OS is about to execute the specific instruction address. The event record contains the instruction address. We use a debug register to cause a VM exit and for the VMM to obtain the control. The IA-32 architecture has hardware debug facilities that can cause a debug exception at the execution of the specific instruction address. The address is specified in a debug register. For Intel VT-x, the exception bitmap defines which exception causes a VM exit. By setting the exception bitmap appropriately, a debug exception causes a VM exit; thus, the VMM obtains the control.

At Step 3, we use the single-step execution mode, and do not use a debug register. It is a special mode for debugging. It executes only a single instruction and causes a debug exception; thus, by using this mode, the VMM obtains the control after the single-step execution.

5 Current Status and Experiment Results

This section describes the current status and the experiment results. We implemented the proposed logging and replaying mechanism in our VMM. The Linux operating system can can boot and also be replayed on the VMM. All experiments described below

Table 1. Source Line of Code to Support the Logging and Replaying Mechanism in VMM

Part	SLOC	Ratio [%]
VMM without logging and replay	9,099	65.4
SMP Support	3,123	22.4
Logging and replaying mechanism	1,694	12.2
Total	13,916	100

Table 2. Time to Boot Linux with and without Logging and Replaying

	Time to boot Linux [sec]	Ratio [%]
No logging	2.284	1
Primary with logging	2.286	100.1
Backup with replaying	2.319	101.5

were performed on the Dell Precision 490 system, which is equipped with Xeon 5130 2.00GHz CPU and 1 GB memory. The version of the Linux kernel is 2.6.23.

First, we show the implementation cost of the logging and replaying mechanism. Second, we show the overheads to boot the Linux. Finally, we show the size of the log and a breakdown of the logged events, and discuss a way to reduce the log size.

5.1 Implementation Cost

The logging and replaying mechanism described in this paper was implemented in our own VMM. Table 1 shows the source lines of code (SLOC) including empty lines and comments. In order to support an SMP environment and to run two Linux instances, one for the primary and another for the backup, 3,123 SLOC were added. They include the code for the initialization of the secondary core, SMP related devices, such as Local APIC and I/O APIC, the modifications to shadow paging, and the shared memory. In order to support the logging and replaying mechanism, 1694 SLOC were added. They include the code for the logging, the ring buffer, and the replaying. In total, SLOC increased 52.9 % in order to realize the proposed system architecture.

5.2 Boot Time Overheads

In order to evaluate the overheads of the logging and replaying, we first measured the boot time of the Linux. The time measured is from `startup_32`, which is the beginning of the the Linux kernel's boot sequence, and to the point, where the init script invokes `sh`, which is the user level shell program. The measurements were performed several times, and their averages are shown as the results.

Table 2 shows the results of the measurements. The table shows the actual boot times in seconds and also the ratio relative to the boot time of the original Linux. The logging imposes almost no overhead that is only 0.1 %[1], and the replaying imposes 1.5 % of the overhead.

[1] It is, actually, slightly less than 0.1 % but is rounded to it.

Table 3. Benchmark Results with and without Logging and Replaying

	fork [mili sec]	fork+exec [mili sec]
No logging	1.336	2.087
Primary with logging	1.357	2.153
Backup with replaying	1.393	2.462

The logging and replaying impose 2 mili seconds and 35 mili seconds of the overhead, respectively. The logging requires VM exits everywhere events need to be recorded while the replaying also requires VM exits everywhere events may need to be replayed. The replaying imposes the more overhead than the logging because the replaying requires the VMM to point out the exact addresses and timings where events need to be replayed. As described in Section 3, loops and repeating instructions make that pointing more difficult because the same instruction addresses are executed multiple times; thus, the difference of the overheads of the logging and the replaying is caused by the cost to point out the exact addresses and timings where events need to be replayed.

5.3 Benchmark Overheads

We further performed the measurements using the two benchmark programs. One benchmark program measures the cost of the fork system call, and the other one measures the combined cost of the fork and exec system calls. They basically perform the same as those included in the LMbench benchmark suite [7], while they are modified to use the RDTSC instruction in order to measure times. The measurements were performed 1000 times, and their averages are shown as the results.

Table 3 show the results of the measurements. The results show that the overheads of the logging are small for both the fork and fork+excec system calls. The overhead of the replaying for the fork+exec system call is, however, higher. In order to further examine the source of the overhead, we calculate the costs of the exec system call from the measured values. The calculated costs of the exec system call from Table 3 is 0.751, 0.796, and 1.069 mili seconds for no logging, primary with logging, and backup with replaying, respectively. The ratios are 106%, 142%, and 134% for primary per no logging, backup per no logging, and backup per primary, respectively. Since the use of only the exec system call is not typical, those calculated overheads will directly impact overall system performance. They rather give the insights for the analysis of the overheads.

The fork and exec system calls behave quite a lot differently as they provide the totally different functions. The fork system call creates a copy of the calling process. It needs to allocate an in-kernel data structure that represents a new process, while it can reuse the most of the other memory images for a new process by the copy-on-write technique. The exec system call, on the other hand, frees the most of memory images except for the in-kernel data structure of the calling process. It then allocates necessary memory regions in order to load an newly executing program. Such loading causes the numerous times of data copying, and external interrupt events during the data copying instructions are expensive to replay. It is very likely to be a reason why the overhead to replay the exec system call is high.

Table 4. Breakdown of Logged Events

Event	Count	Ratio [%]
Timer interrupts	207.5	8.00
Serial line interrupts	2	0.08
IN instruction	2385	91.93
Total	2594.5	100.01

5.4 Size of Log and Breakdown of Logged Events

Table 4 shows a breakdown of the logged events while the Linux was booted.[2] From the ratios of a breakdown, we can see that the most of the logged events are IN instruction. The device driver of the serial line often uses IN instruction in order to examine if the device is ready to transmit data so that data can be written into it. The drivers of the other devices, such as PIC (Programmable Interrupt Controller) and PIT (Programmable Interval Timer), also use IN instruction. Especially, PIC is manipulated every time an interrupt handler is invoked in order to acknowledge an EOI (End of Interrupt handling) and also to mask and umask the corresponding IRQ#.

From the number of the total events, the size of the log after booting the Linux is calculated at 83.02 KB. 83.02 KB of the log is produced for 2.286 seconds of the boot time, meaning 36.32 KB of the log for every second; thus, by using 4MB of the log buffer, the execution of the backup can be deferred for 112.8 seconds if the same production rate of the log is assumed.

6 Discussion for Improvements

This section discusses possible improvements for the described logging and replaying mechanism. There are two obvious issues, one is the log size and the other is the overheads. We discuss them in the rest of this section.

6.1 Reduction of Log Size

We discuss several ways to reduce the log size in order to make the deferred time longer. A straightforward way to do it is by reducing the size of each event. Currently, the fixed size of 32 byte is used to record each event. 32 byte was chosen because it is well aligned with the IA-32 processor's cache line size of 64 byte.

In order to reduce the record size, we can change the data size for the different event types. In other words, by employing the exact size for each event type, the log size can be reduced. The record size of an IN instruction event can be reduced to 2 bytes, which consist of 1 byte for the event type and 1 byte for input data. The record size of an interrupt event can be reduced to 15 bytes, which consist of 1 byte for the event type, 1 byte for IRQ#, 4 byte for the instruction address, 5 byte for the number of branches, and 4 byte for the value of %ecx register. If we can decode the instruction where an interrupt was injected at the time of the logging, an interrupt event can be divided into two types, one with the value of %ecx register and the other without it.

[2] The total of the ratios is 100.01 %, which shows a round error of 0.01 %.

By using the record sizes of 2 byte for an IN instruction event and 15 byte for an interrupt event, the size of the log after booting the Linux can be reduced to 7.91 KB. It is only 9.5 % of the original log size; thus, the significant reduction of the log size is possible. By using these record sizes, 4MB of the log buffer can defer the execution of the backup for 1183.7 seconds (19 minutes and 43.7 seconds).

Another way is that the VMM emulates a serial device in a specific way to reduce the log size. The serial line device driver of the Linux kernel executes a loop to wait until the device becomes ready to transmit data by examining the status of the device issuing IN instruction. Instead, the VMM can always tell the Linux kernel that the device is ready to transmit data, and executes a loop to wait until the device becomes ready to transmit data. In this way, the Linux kernel does not need to issue a number of IN instructions, so that the events that need to be logged can be reduced.

6.2 Reduction of Overheads

We focus on reducing the replaying overhead since the replaying overhead is larger and the logging overhead is relatively small. The benchmark results described in Section 5.3 revealed replaying the exec system call is expensive, and the high overhead of replaying external interrupt events during the data copying instructions is a possible reason. It is so because REP instructions are commonly used for the data copying since it provides the most efficient way to copy data from one place to another. A REP instruction followed by the MOVS instruction copies the data pointed by %esi register to the region specified by %edi register using %ecx register as its counter. It is a single instruction but repeats the execution of the MOVS instruction for the times specified by %ecx. An interrupt is serviced between the executions of the MOVS instruction within the single instruction pointer address. Thus, the only way to pinpoint the timing to deliver an external interrupt within the REP instruction is to execute the instruction in the single step mode until the value of %ecx register becomes equal to the logged value. The execution in the single step mode causes a VM exit after the execution of every MOVS instruction within a REP instruction. The VMM checks the value of %ecx register. If the value is not equal to the logged one, it resumes in the single step mode. If the value becomes equal to the logged one, the VMM cancels the single step mode and injects the logged external interrupt. Therefore, the overhead of replaying a REP instruction becomes larger as its execution in the single step mode is longer.

From the above reasoning, one straightforward way to reduce the replaying overhead is avoid the use of the single step mode. It can be done by dividing the execution of a REP instruction into the two parts, pre-injection and post-injection. The pre-injection and post-injection parts are the execution before and after the injection of an external interrupt, respectively. By adjusting the value of %ecx register and setting the breakpoint after the corresponding REP instruction, the execution of the pre-injection part finishes without a VM exit at every MOVS instruction within a REP instruction. The breakpoint is taken after the execution of the pre-injection part. Then, the VMM injects an external interrupt. After handling the injected external interrupt, the post-injection part is executed. It works because the value of %ecx register is not used by a MOVS instruction within a REP instruction. This way significantly reduces the number of VM exits; thus, we can expect the reduction of the replaying overhead by a certain amount.

Avoiding the use of the single step mode is also possible by a device on the primary at the time of logging. When an external interrupt is delivered, the VMM first takes it and examines the current instruction. If the current instruction is not REP, the VMM injects the interrupt. If the current instruction is REP, the VMM sets the breakpoint at the next instruction and resumes the execution. After the execution of the REP instruction, the VMM injects the interrupt. This way is much simpler, but the delivery of interrupts can be delayed.

7 Related Work

There are several other studies on the logging and relaying mechanisms in VMMs. We took a similar approach to ReVirt [4], Takeuchi's Lightweight Virtual Machine Monitor [10], and Aftersight [11] in terms of the basic mechanisms. PMC is effectively used to count the number of events in their work and ours. They, however, store the log in storage devices, and perform the replaying later. We propose the system that the both logging and replaying are executed simultaneously but with some time difference on the same machine. The details of the mechanism and implementation of our work are also different from them. ReVirt employs UMLinux as its VMM, which does not use Intel VT-x and emulates devices. Aftersight perform the replaying on QEMU system emulator but not a virtual machine on a VMM. Takeuchi's Lightweight Virtual Machine Monitor uses Intel VT-x but only supports a simple RTOS without a memory protection feature by MMU.

Bressoud's fault tolerant system [1] uses a different mechanism from ours to enable the logging and relaying. It performs the logging and the replaying periodically while our mechanism performs them on demand. The performance of Bressoud's system heavily depends of the timing parameter that defines the period for the logging and the replaying; thus, it does not perform well for interactive uses. It is different from ours also in terms of the system configuration that the primary and the backup are implemented on different machines connected with each other by network.

8 Summary

We proposed a system that enables the complete tracing of system failures and such execution traces are used as their evidences. We employ two virtual machines, one for the primary execution and the other for the backup execution. The backup virtual machine maintains the past state of the primary virtual machine along with the log to make the backup the same state as the primary. When a system failure occurs on the primary virtual machine, the VMM saves the backup state and the log. By replaying the backup virtual machine from the saved state following the saved log, the execution path to the failure can be completely analyzed and can provide a concrete evidence.

We developed such a logging and replaying feature in a VMM. The VMM is developed from scratch to run on an SMP PC compatible system. It can log and replay the execution of the Linux operating system. The experiments show that the overhead of the primary execution is only fractional, and the overhead of the replaying execution to boot up the backup is less than 2%.

References

1. Bressoud, T., Schneider, F.: Hypervisor-Based Fault Tolerance. ACM Transactions on Computer Systems 14(1), 80–107 (1996)
2. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. In: Proceedings of the 19th ACM Symposium on Operating System Principles, pp. 164–177 (October 2003)
3. Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., Warfield, A.: Remus: High Availability via Asynchronous Virtual Machine Replication. In: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (2008)
4. Dunlap, G., King, S., Basrai, M., Chen, P.: ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In: Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation, pp. 211–224 (2002)
5. Goldberg, R.P.: Survey of Virtual Machine Research. IEEE Computer, 34–45 (June 1974)
6. Intel Corporation. IA-32 Intel Architecture Software Developer's Manual
7. McVoy, L., Staelin, C.: LMbench: Portable Tools for Performance Analysis. In: Proceedings of the USENIX Annual Technical Conference, pp. 279–294 (January 1996)
8. Neiger, G., Santoni, A., Leung, F., Rodgers, D., Uhlig, R.: Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. Technical Report, Intel Corporation (2006)
9. Rosenblum, M., Garfinkel, T.: Virtual Machine Monitors: Current Technology and Future Trends. IEEE Computer, 39–47 (May 2005)
10. Takeuchi, S., Sakamura, K.: Logging and Replay Method for OS Debugger Using Lightweight Virtual Machine Monitor. IPSJ Journal 50(1), 394–408 (2009)
11. Chow, J., Garfinkel, T., Chen, P.: Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In: Proceedings of the USENIX 2008 Annual Technical Conference, pp. 1–14 (June 2008)

Code Generation for Embedded Java with Ptolemy

Martin Schoeberl, Christopher Brooks, and Edward A. Lee

UC Berkeley, Berkeley, CA, USA
{mschoebe,cxh,eal}@eecs.berkeley.edu

Abstract. Code generation from models is the ultimate goal of model-based design. For real-time systems the generated code must be analyzable for the worst-case execution time (WCET). In this paper we evaluate Java code generation from Ptolemy II for embedded real-time Java. The target system is the time-predictable Java processor JOP. The quality of the generated code is verified by WCET analysis for the target platform. Our results indicate that code generated from synchronous data-flow and finite state machine models is WCET analyzable and the generated code leads to tight WCET bounds.

1 Introduction

In this paper we investigate Java code generation from a model-based design. Specifically we use Ptolemy II [4] for the modeling and code generation and target the embedded Java platform JOP [12]. Ptolemy II is extended with run-time modeling actors to represent low-level I/O operations of the target platform in the simulation environment. The Ptolemy II code generation framework is extended with code generating actors to implement the low-level I/O operations.

The target platform includes a worst-case execution time (WCET) analysis tool. Therefore, we are able to analyze the WCET of various actors provided by Ptolemy II and the WCET of the generated application. Comparing WCET analysis with measured execution time on the target shows that code generated from Ptolemy II models results in Java code where a tight WCET bound can be derived. WCET analysis also reveals that modeling of applications without considering the execution time (e.g., using double data type when not necessary) can result in surprisingly high execution time.

The paper is organized as follows: in the remainder of this section we give some background information on Ptolemy II and JOP. In Section 2 code generation from models and the implications for low-level I/O operations on an embedded platform are described. Implementation details on the embedded platform with a few example applications is given in Section 3. In Section 4 we show that the generated code is WCET analyzable and usually leads to tight WCET bounds. We discuss our findings in Section 5. The paper is concluded in Section 6.

1.1 Ptolemy II

Ptolemy II [4] is a modeling and simulation tool for heterogenous embedded systems. Ptolemy II allows generation of C and Java code from the models. We use the latter feature to build embedded applications targeted for a Java processor.

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 155–166, 2010.

Ptolemy II is a Java-based component assembly framework with a graphical user interface called Vergil. The Ptolemy project studies modeling, simulation, and design of concurrent, real-time, embedded systems. The principle in the project is the use of well-defined models of computation that give the individual models execution semantics.

In Ptolemy II, models of computations are also known as domains. In this paper we focus on the synchronous dataflow (SDF) domain [9] combined with modal models [5]. The Giotto [7] and the synchronous/reactive [3] domain are also interesting code generation targets and we assume they would work with the presented infrastructure. As target platform we have chosen an embedded real-time Java processor.

The building blocks of a Ptolemy II model are actors. Actors exchange data by passing tokens between ports that connect actors. The actual execution semantics of connected actors is defined by the model of computation or execution domain. In our evaluation, the synchronous data-flow (SDF) domain is explored. SDF handles computations on streams. An SDF model constructs a network of actors that are then scheduled for execution. The schedule of the firings can be computed statically [9], making this domain an easy target for generation of efficient code.

The modal model [5] domain adds finite state machines (FSM) to the computing power of the SDF domain. Modal models can represent simple FSMs or provide an abstraction of mode changes. In the later case the FSM represents different modes in the applications and the states are refined by submodels for each mode.

1.2 The Java Processor JOP

We target embedded Java systems with our code generation framework. As a first example we use JOP [12], a Java processor especially designed for real-time systems. JOP implements the Java virtual machine (JVM) in hardware. That means bytecode, the instruction set of the JVM, is the instruction set of JOP. The main feature of JOP is the ability to perform WCET analysis at bytecode level. This feature, and a special form of instruction caching, simplify the low-level part of the WCET analysis. Furthermore, performing WCET analysis at the bytecode level is simpler than performing it at the executable level for compiled C/C++ programs. In Java class files more information is available, e.g., receiver types, than in a *normal* linked executable.

The runtime environment of JOP is optimized for small embedded applications with real-time constraints. JOP includes a minimal version of the Java library (JDK) and provides a priority based thread scheduler. Real-time threads on JOP are either periodic (time-triggered) or software driven event triggered. Hardware events can trigger an interrupt handler. The whole system, including the scheduler and interrupt handlers, is written in Java. In the context of code generated from Ptolemy II we use periodic threads to execute the models.

2 Code Generation from Models

Code generation for embedded systems is challenging due to several constraints. Embedded systems have severe resource limitations in processing power and in memory size. Although Java is known to have quite a large memory footprint (counted in MB),

JOP's memory footprint is in the range of several KB. As a consequence, only a small part of the Java library is available. Code generated from Ptolemy II models is conservative enough to target small embedded Java systems.

2.1 Code Generation with Ptolemy II

Within Ptolemy II several approaches for code generation have been evaluated.

Copernicus: A code generation framework that included "deep code generation", where Java bytecode of the actors is analyzed and specialized for use in the particular model [11]. Copernicus deep code generation generated code for non-hierarchical, synchronous dataflow (SDF) models. JHDL [10] used Copernicus to generate FPGA code. However, one drawback of the deep code generation approach is that it is affected by changes to the Ptolemy II runtime kernel.

Codegen: A template based approach [20] where we define adapter classes associated with Ptolemy II actors. The adapter classes define the C or Java implementation or the corresponding Ptolemy II actor. The code generator performs type and width inference on the model and generates optimized code.

CG: Another template based approach that is currently under development. The CG effort is focused on composite code generation [19], where reusable code is generated for each group of actors. Composite code generation will allow model developers to modify the model and generate code only for changed portions. Composite code generation also allows generation of Java code for very large models.

For the experiments targeting JOP we use the Codegen and the CG code generation frameworks. Support for the low-level I/O with JOP has been added to both systems.

2.2 Runtime Support on JOP

Code generated for the SDF model of computation is usually executed periodically. The period is defined in the SDF director of the model. On JOP we leverage the available real-time scheduler and execute the SDF model in a periodic thread. Multiple periodic threads can be used to combine the SDF model with other code, written in Java or generated from a model. To reason about the timing properties of the whole system, all (periodic) threads need to be WCET analyzable. Given the periods and the execution time, schedulability analysis gives an answer if all deadlines will be met.

If the application consists of just an SDF model, then a simple, single threaded approach, similar to a cyclic executive, is also possible. The release to execute an iteration of the SDF model can be synchronized via an on-chip clock. At the end of one iteration the processor performs a busy wait, polling the clock, until the next release time. Cycle accurate periods can be obtained by performing this wait in hardware. We have implemented this hardware support, called *deadline instruction*, in JOP [16].

As Ptolemy II is mainly used for simulation, it lacks actors for interfacing low-level I/O. Within JOP we use so called hardware objects [14] to represent I/O ports as standard Java objects. We have implemented input and output actors for several I/O devices on JOP. Details on the I/O actors are given in the next subsections.

Listing 1.1. Implementation of a low-level output actor

```
public class JopSerialWrite extends Sink {

    public JopSerialWrite(...) {}

    private int lastVal;
    private int val;

    // Save the value for the output port.
    public void fire() {
        super.fire();
        lastVal = val;
        if (input.hasToken(0)) {
            lastVal =
                ((IntToken) input.get(0)).intValue();
        }
    }

    // Write the saved value to the output port.
    public boolean postfire() {
        val = lastVal;
        writePort(val);
        return super.postfire();
    }
}
```

2.3 Semantics of an Actor

Actors are the components in Ptolemy II that represent computation, input, and output. Details of actor semantics can be found in [10]. The two methods central to an actor are `fire()` and `postfire()`: The `fire()` method is the main execution method. It reads input tokens and produces output tokens. In some domains, `fire()` is invoked multiple times, e.g., to find a fixpoint solution in the synchronous/reactive domain. The `postfire()` method is executed once and shall update persistent state. The main points an actor developer has to consider are when to calculate the output (`fire()`) and when to update state (`postfire()`).

2.4 Input/Output Actors

An I/O operation often updates persistent state, e.g., output of a character to a terminal updates persistent state. Therefore, `fire()` is not the method where the I/O operation shall happen. For output the output data (token) needs to be saved in `fire()`, but the actual output action has to happen in `postfire()`.

It is less obvious, but reading an input value can also change the state of the system. As an example, reading from the input buffer of a communication channel (the serial line), consumes that value. Therefore, the actual read operation has to happen exactly once. The input value is read in the first in invocation of `fire()` per iteration and saved in a local variable. The output token for the read actor uses the value of that local variable on each invocation in one iteration. The start of a new model iteration is marked within the `postfire()` method and an actual I/O read is performed at the next `fire()`.

In the following we give examples of I/O actors for the serial port on JOP. For the access to the device registers we use *hardware objects* that represent I/O devices as plain

Listing 1.2. Java code template for a low-level output actor

```
/***preinitBlock***/
int $actorSymbol(val) = 0;
int $actorSymbol(lastVal);
static com.jopdesign.io.SerialPort
    $actorSymbol(ser) =
    com.jopdesign.io.IOFactory.
        getFactory().getSerialPort();
/**/

/*** fireBlock($channel) ***/
$actorSymbol(lastVal) = $actorSymbol(val);
// no check on available input token here
// so above assignment is useless
$actorSymbol(lastVal) = $get(input#$channel);
/**/

/*** postfireBlock ***/
$actorSymbol(val) = $actorSymbol(lastVal);
if (($actorSymbol(ser).status &
    com.jopdesign.io.SerialPort.MASK_TDRE)!=0) {

    $actorSymbol(ser).data = $actorSymbol(val);
}
/**/
```

Java objects [15]. The hardware objects are platform specific as they are representing platform specific I/O devices. However, the hardware object that represents a serial port interface, used in this section, has been used without alteration in five different JVMs (CACAO, OVM, SimpleRTJ, Kaffe, and JOP [14]).

Output Actor. Listing 1.1 shows the anatomy of a low-level output actor. The code shows how the output actor is implemented for the simulation mode in Ptolemy II. Within `fire()` an input token is read and stored in an actor private variable. The actual output, the state changing action, is performed in `postfire()`, which is guaranteed to be executed only once and after `fire()`.

The code generator can remove most of the overhead involved in methods calls. Therefore, the code template, shown in Listing 1.2, contains only the core logic. Block markers, within the `/*** ***/` style comment, delimit a code block. Macros (symbols starting with $) in the code template are substituted by the code generator. The macro `$actorSymbol(val)` generates a static variable with a unique name for the actor local variable `val`. `$get(input#$channel)` reads a token from a channel of the input port. Actor parameters, attributes assigned at design time, can be accessed with `$param(name)`.

The code example shows the usage of a hardware object to access the serial port. The hardware object, which represents the serial port registers, is *created* by a factory method. As hardware, and the representing objects, cannot be just created with `new`, a system specific factory is used to return an instance of a hardware object. In the `postfireBlock` the device is accessed via this hardware object.

Listing 1.3. Implementation of a low-level input actor

```
public class JopSerialRead extends Source {

    public JopSerialRead(...) {}

    private boolean firstFire = true;
    private IntToken val = new IntToken(0);

    // Read the input port on the first
    // invocation of fire(). Send the value
    // on each fire().
    public void fire() {
        super.fire();
        if (firstFire) {
            int v = readSerialPort();
            val = new IntToken(v);
            firstFire = false;
        }
        output.send(0, val);
    }

    // Enable read for the next fire().
    public boolean postfire() {
        firstFire = true;
        return super.postfire();
    }
}
```

Although the template code looks a little bit verbose, the generated code using this template is quite short. The `fire()` and `postfire()` methods are inlined to avoid the overhead of the method invocation.

Input Actor. Listing 1.3 shows the concept of an input actor. This code example shows how an input actor is implemented for the simulation. An actor private Boolean variable is used to detect the first invocation of `fire()` to perform the actual I/O read function and save the value. On future invocations of `fire()` that value is returned. The flag to indicate the first firing of the actor is reset in `postfire()`. The code template for the input actor is similar to the one shown for the output actor.

3 Implementation

Models are executed on the target platform periodically. The period can be configured in the SDF director. We changed the code generator to output the configured period as a constant. On the target platform the model class is instantiated, initialized, and executed. For the periodic execution we use the periodic real-time threads available in the JOP runtime. The code in Listing 1.4 shows the execution of the model.
For the periodic execution of the generated model we had to change the Java code generator to provide access to the period of the SDF director. If the period of the SDF director is set to 0 the model is executed in a tight loop – in fact it is running at the maximum possible frequency.[1]

[1] In the implementation all values less than 100 microseconds are treated as period 0.

Listing 1.4. Execution of the model in a periodic thread

```
final Model model = new Model();
model.initialize();

int us = (int) (model.PERIOD * 1000000);
// If there is a useful period, run it in a periodic thread.
// If not, just in a tight loop.
if (us >= 100) {
    new RtThread(1, us) {
        public void run() {
            for (;;) {
                try {
                    model.run();
                } catch (Exception e) {
                    e.printStackTrace();
                }
                waitForNextPeriod();
    }};
    RtThread.startMission();
} else {
    for (;;) {
        try {
            model.run();
        } catch (Exception e) {
            e.printStackTrace();
}}}
```

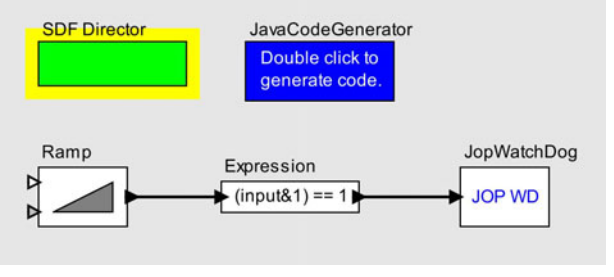


Fig. 1. A simple Ptolemy II model generating the watchdog trigger

3.1 Embedded Hello World

The first low-level I/O actor we implemented was the interface to the watchdog LED. With output to a single LED the embedded version of Hello World – a blinking LED – can be designed with Ptolemy II. Figure 1 shows a simple model that toggles the watchdog LED. The rightmost block is a sink block that sets the LED according to the received Boolean token. With a Ramp block and an Expression block the LED blinks at half of the frequency of the SDF director when executing the generated code on the target.

3.2 An SDF Example with a State Machine

Besides data flow actors that are driven by the synchronous data flow director, finite state machines (FSM) are needed to build meaningful embedded applications. We have ported the FSM code generator for C to the Java based code generation.

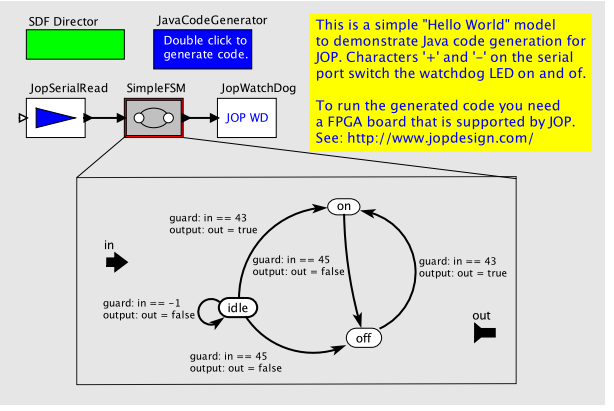


Fig. 2. A Ptolemy II model with a serial interface, an FSM, and the watchdog LED with the inner FSM shown

Figure 2 shows an example containing all ingredients to build embedded programs with Ptolemy II for JOP. The serial port actor reads commands from a host and forwards them to the state machine, which itself generates the output for the watchdog LED. The LED can be turned on with a '+' and turned off with a '-' character received on the serial line.

3.3 Lego Robot

As an example for an application we have modeled a line following robot. The hardware for this experiment is a LEGO Mindstorms robot with an interface of a JOP based FPGA board to the LEGO sensors and actuators. The interface contains several analog inputs, switch inputs, and pulse width modulated (PWM) output for the motors. Furthermore, the motor interface provides an input to read the actual speed of the motors. During the off cycles of the PWM the motor acts as generator and the produced voltage (also called back EMV), which is proportional to the actual revolving speed, can be read. The application periodically samples the input of the light sensor, calculates the control law, and puts out the drive values for the two motors.

4 WCET Analysis

JOP provides a simple low-level timing model for WCET analysis. As almost all Java bytecodes execute in constant time, the WCET analysis can be performed at bytecode level. Due to this simplification, several WCET analysis projects target JOP [6,8,17].

For our experiments we use the WCET analysis tool WCA [18], which is part of the JOP distribution. WCA implements two modes of WCET analysis: the implicit path enumeration technique (IPET) and WCET analysis based on model checking [8]. The results presented in this paper are based on the faster IPET mode of WCA.

Static WCET analysis gives an upper bound on the execution time of a program on a specific target platform. Programs are WCET analyzable when all loops and the recursion depths are bound. It is usually impossible to derive the *real* WCET due to two factors: (1) the analysis has not enough knowledge of impossible execution paths and (2) the processor cannot be modeled in full detail. The first issue is usually attacked by annotations in the source code of possible paths, whereas the second issue is solved by a simpler, but safe, model of the hardware. Within the JOP project we try to minimize the overestimation of the WCET bounds by providing an execution pipeline that is simple enough to be modeled for the analysis. However, it has to be noted that WCET analysis will almost always (except for trivial programs) give a conservative estimate, the WCET bound. Furthermore, when comparing WCET values given by the analysis with measurements one has to keep in mind that the measurement usually will not provide the real WCET value. If this would be the case, we would not need to statically analyze the program.

4.1 Model Examples

We have evaluated different models with WCET analysis and execution time measurements on the target. Table 1 shows the analyzed and measured execution time in clock cycles. The `WatchDog` example is the embedded hello world model, as shown in Figure 1. With such a simple model the measured execution time is almost the same as the analyzed time. The WCET for the FSM example `SimpleFSM`, which is the model from Figure 2, contains more control code and during the measurement the data dependent WCET path was not triggered. This model is an example where simple measurement does not give the correct WCET value.

Table 1. WCET bounds and measured execution time for different models on JOP

Model	WCET (cycles)	Measured (cycles)
`WatchDog`	277	244
`SimpleFSM`	9539	3843
`Filter`	1818	1734
`Follower`	3112	2997
`Add (int)`	656	604
`Add (double)`	19545	12092

The third application, `Filter`, represents a digital signal processing task with a finite impulse response filter. The source for the filter is a pulse generating actor. As the generated code is basically straight line code, the measured execution time and the WCET bound are almost identical.

The `Follower` example is the line following LEGO robot. Again, the generated code is easy to analyze and the WCET bound is tight. From the examples without data dependent control flow we see that the WCET bound is close to the measured execution time.

4.2 Data Types

The four examples use integer data types in the models. For an embedded system without a floating point unit this is the preferable coding style. JOP does not contain a floating point unit (FPU).[2] Floating point operations on JOP are supported via a software emulation of the floating point bytecodes. To demonstrate how expensive double operations (the default floating point type in Java) are, we compare a model with two ramp sources, an adder, and a sink actor for 32-bit integer data with the same model for 64-bit double data. In Ptolemy II the models are identical. Only the port type needs to be changed for the requested data type. The code generation framework then automatically instantiates the correct type of the actor. The example model (`Add (double)`) with double data type executes in the worst case about 30 times slower than the integer version. This illustrates the importance of choosing the right data type for the computation on an embedded platform. A developer, educated on standard PCs as computing platform is usually not aware of this issue, which is typical for a resource constraint embedded platform. Using WCET analysis, right from the beginning of the development, is important for economic usage of computing resources.

5 Discussion

The integration of JOP into Ptolemy II was relative straightforward. One of the authors had no real knowledge on the usage of Ptolemy II in general and the code base specifically. The documentation of Ptolemy II [2] and the source code are well organized. Getting used to design a few simple test models was a matter of hours. Adding new actors in the simulation and in the Java code generation base took less than two days, including familiarizing with the Ptolemy II sources.

On the JOP side we had to update the provided library (JDK) to better support double data types. As double operations are very expensive on JOP, their support was quite limited.

5.1 Low-Level I/O

Java, as a platform independent language and runtime system, does not support direct access to low-level I/O devices. One option to access I/O registers directly is to access them via C functions using the Java native interface. Another option is to use so called hardware objects [14], which represent I/O devices as plain Java objects. The hardware objects are platform specific (as I/O devices are), but the mechanism to represent I/O devices as Java objects can be implemented in any JVM. Hardware objects have been implemented so far in five different JVMs: CACAO, OVM, SimpleRTJ, Kaffe, and JOP. Three of them are running on a standard PC, one on a microcontroller and one is a Java processor. It happened that the register layout of the serial part was the same for all three hardware platforms. Therefore, the hardware object for the serial device, and the serial port actor, can be reused for all five JVMs without any change.

[2] An FPU for 32-bit float is available in the source distribution. As that unit is twice as large as JOP, the FPU is disabled per default.

5.2 Memory Management

One of the most powerful features of Java is automatic memory management with a garbage collector. However, the runtime implications of garbage collection in a real-time setting are far from trivial. The real-time specification for Java [1] provides scoped memory as a form of predictable dynamic memory management. The actors we used in our examples do not allocate new objects at runtime. For more complex actors, which use temporary generated objects, scoped memory can be used.

Another source of created objects are the tokens for the communication between the actors. In our examples we used only primitive data types and the code generation framework maps those token to primitive Java types. For complex typed tokens a solution for the generated objects need to be found, either using (and trusting) real-time garbage collection or provide a form of token recycling from a pool of tokens.

6 Conclusion

In this paper we have evaluated code generation for embedded Java from Ptolemy II models. The code generation framework can generate C and Java code. For our embedded Java target, the real-time Java processor JOP, we explored the Java code generation. Code generation in Ptolemy is template based and generates simple enough code that is WCET analyzable. The combination of the time-predictable Java processor as target and the generated code from Ptolemy enables WCET analysis of model-based designs. We argue that this combination is a step towards model-based design of hard real-time systems.

Both projects, Ptolemy II and JOP, used in the paper are available in source form; see `http://chess.eecs.berkeley.edu/ptexternal` and `http://www.jopdesign.com/`. The build instructions for JOP are available in Chapter 2 of the JOP Reference Handbook [13]. An example for the execution of code generated from Ptolemy models in a periodic thread can be found in `java/target/src/test/ptolemy/RunIt.java`.

References

1. Bollella, G., Gosling, J., Brosgol, B., Dibble, P., Furr, S., Turnbull, M.: The Real-Time Specification for Java. Java Series. Addison-Wesley, Reading (June 2000)
2. Brooks, C., Lee, E.A., Liu, X., Neuendorffer, S., Zhao, Y., Zheng, H.: Heterogeneous concurrent modeling and design in Java. Technical Report UCB/EECS-2008-28/29/37, University of California at Berkeley (April 2008)
3. Edwards, S.A., Lee, E.A.: The semantics and execution of a synchronous block-diagram language. Science of Computer Programming 48(1) (2003)
4. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the Ptolemy approach. Proceedings of the IEEE 91(2), 127–144 (2003)
5. Girault, A., Lee, B., Lee, E.A.: Hierarchical finite state machines with multiple concurrency models. IEEE Transactions on Computer-aided Design of Integrated Circuits And Systems 18(6), 742–760 (1999)
6. Harmon, T.: Interactive Worst-case Execution Time Analysis of Hard Real-time Systems. PhD thesis, University of California, Irvine (2009)

7. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: A time-triggered language for embedded programming. Proceedings of the IEEE 91(1), 84–99 (2003)
8. Huber, B., Schoeberl, M.: Comparison of implicit path enumeration and model checking based WCET analysis. In: Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis, Dublin, Ireland, pp. 23–34. OCG (July 2009)
9. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proceedings of the IEEE 75(9), 1235–1245 (1987)
10. Lee, E.A., Neuendorffer, S., Wirthlin, M.J.: Actor-oriented design of embedded hardware and software systems. Journal of Circuits, Systems, and Computers 12(3), 231–260 (2003)
11. Neuendorffer, S.A.: Actor-Oriented Metaprogramming. PhD thesis, EECS Department, University of California, Berkeley (January 2005)
12. Schoeberl, M.: A Java processor architecture for embedded real-time systems. Journal of Systems Architecture 54(1-2), 265–286 (2008)
13. Schoeberl, M.: JOP Reference Handbook: Building Embedded Systems with a Java Processor. CreateSpace (August 2009) ISBN 978-1438239699, http://www.jopdesign.com/doc/handbook.pdf
14. Schoeberl, M., Korsholm, S., Kalibera, T., Ravn, A.P.: A hardware abstraction layer in Java. Trans. on Embedded Computing Sys., (accepted 2010)
15. Schoeberl, M., Korsholm, S., Thalinger, C., Ravn, A.P.: Hardware objects for Java. In: Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008), Orlando, Florida, USA, pp. 445–452. IEEE Computer Society, Los Alamitos (May 2008)
16. Schoeberl, M., Patel, H.D., Lee, E.A.: Fun with a deadline instruction. Technical Report UCB/EECS-2009-149, EECS Department, University of California, Berkeley (October 2009)
17. Schoeberl, M., Pedersen, R.: WCET analysis for a Java processor. In: Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006), pp. 202–211. ACM Press, New York (2006)
18. Schoeberl, M., Puffitsch, W., Pedersen, R.U., Huber, B.: Worst-case execution time analysis for a Java processor. Software: Practice and Experience 40(6), 507–542 (2010)
19. Tripakis, S., Bui, D.N., Geilen, M., Rodiers, B., Lee, E.A.: Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs. Technical Report UCB/EECS-2010-52, UC Berkeley (May 2010)
20. Zhou, G., Leung, M.-K., Lee, E.A.: A code generation framework for actor-oriented models with partial evaluation. In: Lee, Y.-H., Kim, H.-N., Kim, J., Park, Y.W., Yang, L.T., Kim, S.W. (eds.) ICESS 2007. LNCS, vol. 4523, pp. 786–799. Springer, Heidelberg (2007)

Specification of Embedded Control Systems Behaviour Using Actor Interface Automata

Christo Angelov, Feng Zhou, and Krzysztof Sierszecki

Mads Clausen Institute for Product Innovation
University of Southern Denmark
Alsion 2, 6400 Soenderborg, Denmark
{angelov,zhou,ksi}@mci.sdu.dk

Abstract. Distributed Timed Multitasking (DTM) is a model of computation describing the operation of hard real-time embedded control systems. With this model, an application is conceived as a network of distributed embedded actors that communicate with one another by exchanging labeled messages (signals), independent of their physical allocation. Input and output signals are exchanged with the controlled plant at precisely specified time instants, which provides for a constant delay from sampling to actuation and the elimination of I/O jitter. The paper presents an operational specification of DTM in terms of actor interface automata, whereby a distributed control system is modeled as a set of communicating interface automata executing distributed transactions. The above modeling technique has implications for system design, since interface automata can be used as design models that can be implemented as application or operating system components. It has also implications for system analysis, since actor interface automata are essentially timed automata that can be used as analysis models in model checking tools and simulation environments.

Keywords: Distributed control systems, component-based design of embedded software, domain-specific frameworks, distributed timed multitasking, interface automata.

1 Introduction

Control-theoretic models of computer control systems assume a synchronous pattern of computer-plant interaction, featuring sampling at a constant frequency and synchronism, i.e. zero delay between sampling and actuation. These assumptions have been adopted in the *perfect* synchronous model of computation used in a number of programming languages and environments, such as LUSTRE, SIGNAL and ESTEREL [1].

In practice, the perfect synchronous model can only be approximated, because the control program has non-zero execution time C and response time R, $R = C + I$, where I denotes the extra time due to interference (preemption) from higher priority tasks, interrupts, etc. However, such an approximation is valid only when the control computer is infinitely fast, or at least much faster than the plant controlled, such that its response time is orders of magnitude smaller than the sampling period.

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 167–178, 2010.

That is obviously not the case when task response time is of the same order as its period. In that case, the synchrony hypothesis is not valid – there is no longer an effective zero delay between sampling and actuation. Consequently, the real system may exhibit a closed-loop behaviour being substantially different from the modeled one. This requires a re-design of the control system taking into account the computational delay. Unfortunately, the task response time R varies with different invocations because the interference due to higher priority tasks I fluctuates with different task phasings. That is why it is impossible to precisely model the influence of the computational delay in the control-theoretic model of the real system.

The variation of response time is ultimately demonstrated as input and output jitter, which is detrimental to control system behaviour. Its effect is substantial for control loops having small sampling time that is comparable to the task response time R. Theoretical and experimental investigations have shown that I/O jitter may result in poor quality of control and even instability [2].

The above problems can be dealt with by adopting a modified model, i.e. the *clocked* synchronous model of computation [3], which is characterized by a *constant*, non-zero delay from sampling to actuation (e.g. one-period delay). That is why it can be easily taken into account in the discrete-time continuous model of the control system (e.g. by using zero-order hold and unit delay blocks). In this way, the behaviour of the real system becomes identical to its modeled behaviour and I/O jitter is eliminated.

This approach is popular among control engineers and has been recently adopted in a number of software frameworks, e.g. Giotto [5], which employs time-triggered tasks executing at harmonic frequencies, whose deadlines are equal to the corresponding periods.

A general solution to the above problems is provided by a particular version of the clocked synchronous model known as Timed Multitasking [4], which can be used with both periodic time-driven, as well as aperiodic event-driven tasks. This model assumes that task I/O drivers are executed atomically at task release/deadline instants, whereas the task itself is executed in a dynamic scheduling environment. In this way, task I/O jitter is effectively eliminated as long as the task comes to an end before its deadline, which is defined to be less than or equal to period. On the other hand, Timed Multitasking provides for a constant delay from sampling to actuation, which can be taken into account in the discrete-time model of the closed-loop control system.

Distributed Timed Multitasking (DTM) is a model of computation developed in the context of the COMDES framework [8]. It extends the original model to sets of application tasks (actors) executing transactions in single-computer or distributed real-time environments. A *denotational* specification of that model is given in [9], where system operation is described by means of composite functions specifying signal transformations from input signals to output signals, taking into account the constant delay from sampling to actuation. However, a precise operational specification is still missing.

The purpose of this paper is to develop an *operational* specification of distributed control system behaviour in terms of interface automata [7]. These can be used to formally specify actor and system behaviour under DTM, which is a prerequisite for deriving accurate analysis models. Interface automata can also be used as design models needed to develop an operating system environment supporting DTM.

The rest of the paper is structured as follows: Section 2 presents an informal introduction of Distributed Timed Multitasking focusing on the behaviour of system actors during the execution of phase-aligned transactions. Section 3 presents interface automata modelling the behaviour and interaction of embedded actors executing distributed transactions with hard deadlines. Section 4 discusses the implementation of interface automata as operating system components incorporated in the event management subsystem of a real-time kernel. Section 5 presents related research. The last section summarizes the main features of the proposed model and its implications.

2 Distributed Timed Multitasking: An Informal Introduction

Distributed Timed Multitasking (DTM) is a model of computation, which has been developed in the context of COMDES – a component-based framework for hard real-time embedded control systems [8]. In this framework, an embedded system is conceived as a network of active objects (actors) that communicate with one another via labelled state messages (signals) encapsulating process variables, such as *speed*, *pressure*, *temperature*, etc. (see e.g. Fig. 1). DTM extends Timed Multitasking to distributed real-time systems in the context of communicating actors and transparent signal-based communication [9].

An actor consists of a signal processing unit (SPU) operating in conjunction with I/O latches, which are composed of input and output signal drivers, respectively [9]. The input latch is used to receive incoming signals and decompose them into local variables that are processed by the SPU. The output latch is used to compose outgoing signals from local variables produced by the SPU and broadcast them to potential receivers. This is accomplished by means of communication primitives that make it possible to transparently broadcast and receive signals, independent of the allocation of sender and receiver actors on network nodes. Physical I/O signals are treated in a similar manner but in this case, the latches are used to exchange physical signals with the environment at precisely specified time instants.

A control actor is mapped onto a real-time task having three parts: *task input*, *task body* and *task output*, implementing the input latch, SPU and output latch, respectively. The task body is executed in a dynamic priority-driven scheduling environment. It is released by the event that triggers the actor for execution, i.e. a periodic timing event, external interrupt or a message arrival event. During execution it may be preempted by other higher-priority tasks running in the same node, and consequently – suffer from I/O jitter.

Task input and output are relatively short pieces of code whose execution time is orders of magnitude smaller than the execution time of the actor task, which is typical for control applications. They are executed atomically in logically zero time, in separation from the task body (split-phase task execution). Specifically, task input is executed when the actor task is released, and task output – when the task deadline arrives (see Fig. 2). Consequently, task I/O jitter is effectively eliminated as long as the task is schedulable and comes to an end before its deadline.

When a deadline is not specified, the task output is executed immediately after the task is finished. That is e.g., the case with the intermediate tasks of phase-aligned transactions, which have to generate output signals as soon as they are computed,

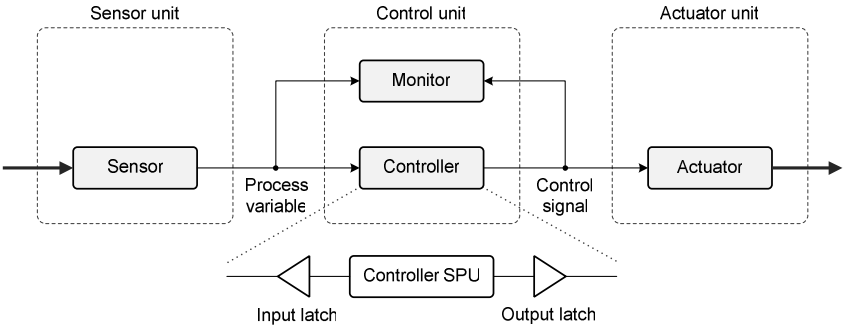


Fig. 1. COMDES model of a distributed embedded system

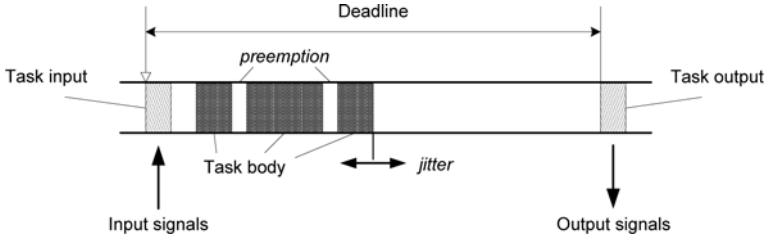


Fig. 2. Split-phase execution of actor under Distributed Timed Multitasking

whereas an end-to-end deadline is imposed on the entire task sequence executing the distributed transaction (see below).

The above techniques can be extended to task sets executing transactions in single-computer or distributed environments. These are treated in the same manner due to the transparent nature of signal-based communication, e.g. the phased-aligned transaction shown in Fig. 3, involving the actors *Sensor* (*S*), *Controller* (*C*) and *Actuator* (*A*) introduced in Fig. 1. This transaction is triggered by a periodic timing event, i.e. a synchronization (*sync*) message denoting the initial instant of the transaction period *T*, with deadline $D \leq T$.

In principle, such transactions suffer from considerable I/O jitter. That is due to task release/termination jitter as well as communication jitter, which is accumulated and ultimately – inherited by the terminal actor. However, in our case input and output signals are generated at transaction start and deadline instants, resulting in constant response time and the effective elimination of I/O jitter.

This resolves the main problem with phase-aligned transactions, which are otherwise simple to implement and commonly used in distributed applications. Transactions involving periodic tasks with the same or harmonic periods are also common for many applications and frameworks, e.g. Giotto [5]. In such transactions task deadlines are usually equal to task periods.

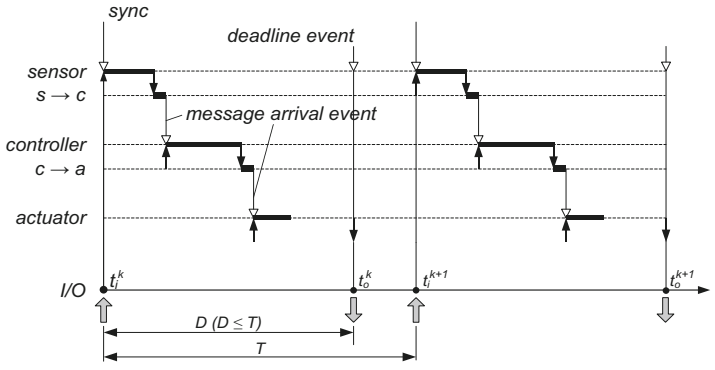


Fig. 3. Jitter-free execution of distributed transactions

The DTM model of computation is presently supported by the HARTEXμ kernel [10]. It has been experimentally validated in a number of computer control experiments, involving physical and computer models of plants such as DC motor, production cell, steam boiler, turntable, etc., as well as an industrial case study – a medical ventilator control system featuring fast, jitter-sensitive control loops with millisecond-range periods and deadlines.

3 Modelling Actor Behaviour with Interface Automata

Interface automata are state machine models used to describe the timed input/output behaviour of software components [7]. Specifically, they can be used to precisely describe the interaction between components in terms of communication protocols implemented by the interfaces of the interacting components, e.g. sequences of method invocations of the corresponding operational interfaces, or message exchanges involving interacting port-based interfaces, etc. In the context of DTM, interface automata can be used to formally specify actor behaviour and interaction, i.e. the behaviour of system actors executing a distributed transaction and communicating with one another via labelled state messages (signals).

The interface automaton can be viewed as an operational component which conducts the execution of a system actor. Accordingly, the actor can be modeled as a composition of input latch, signal processing unit (SPU) and output latch, which are controlled by the interface automaton (see Fig. 4). This model emphasizes separation of concerns: the SPU implements the functional behaviour of the actor in separation from its timed I/O behaviour, which is modeled by the interface automaton.

The interface automaton is enabled for execution by an external *start* event generated by another actor, and is subsequently triggered by periodically arriving tick events that are also used to update a timer measuring the corresponding period and/or deadline intervals. It generates control signals *get input* and *start* in order to activate the input latch and then start the SPU, which generates a *ready* signal when the computation is finished. Finally, the interface automaton generates the signal *produce output* in order to activate the output latch and generate the corresponding output signals. In case of deadline violation, it generates the corresponding exception signal.

This kind of behaviour can be formally specified in terms of periodically executed, event-driven Mealy machines whose transitions are labeled with the corresponding < transition trigger/control signal > pairs. It is possible to define various types of actor interface automata, depending on the start event used to enable the actor and the type of transaction it is involved in. The following discussion assumes application scenarios featuring periodic tasks with harmonic periods or phase-aligned transactions that are common for distributed embedded systems. These can be implemented with several kinds of actor, as follows:

- Periodic time-driven actor with deadline less than or equal to period
- Event-driven actor triggered by either an external event or a message arrival event (e.g. a sync message, application message) – with or without deadline
- Event-driven actor triggered by a message arrival event – with deadline inherited from the transaction deadline (terminal transaction actor)

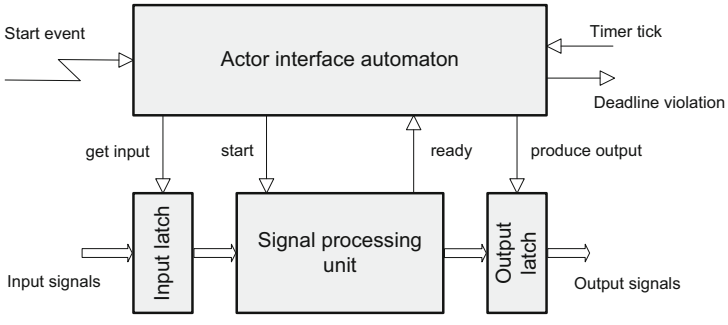
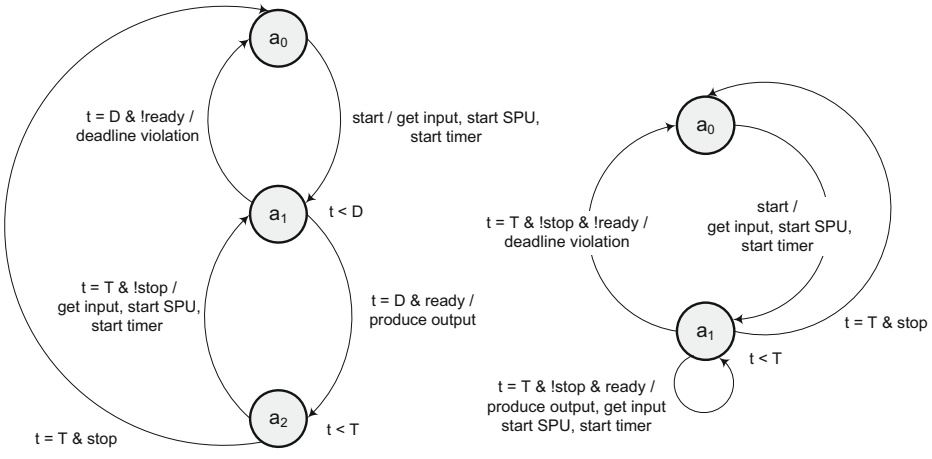


Fig. 4. Control actor modelled as a composition of signal-processing components controlled by an interface automaton

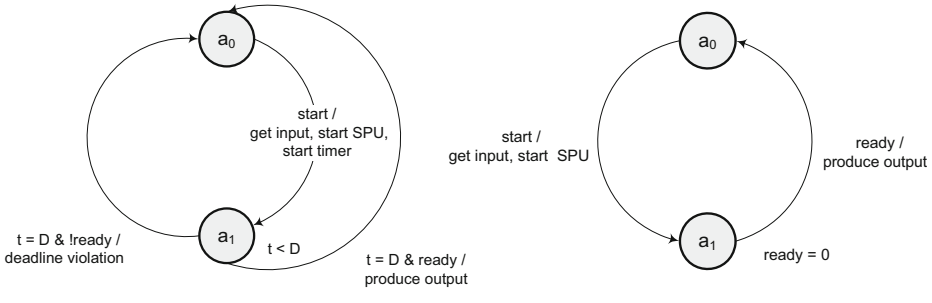
The interface automaton of a periodic time-driven actor with deadline less than period is shown in Fig 5-a. When started, the interface automaton generates the control signals needed to get input signals and start the SPU, starts a timer that will be used to measure both deadline and period, and makes a transition to state a_1 which will be maintained while current time is less than deadline. When $t = D$, a control signal is generated in order to produce output signals, provided that the SPU has finished computation (*ready = 1*), and a transition to state a_2 is made, which will be maintained while current time is less than period. When $t = T$, the interface automaton generates the signals needed to get input and start the SPU, restarts the timer and goes back to state a_1 (waiting until $t = D$). This mode of operation will be repeated over and over again up until a *stop* signal is issued by an external actor that will force a transition back to the initial state a_0. A transition to that state is also enforced in the case of deadline violation, when the interface automaton is in state a_1 and $t = D$ but the SPU has not finished computation (*ready = 0*).



a) with deadline less than period b) with deadline equal to period

Fig. 5. Interface automata of time-driven actors

Fig. 5-b shows the interface automaton of a periodic time-driven actor with deadline equal to period. Such actors are typically used in control applications featuring one-period delay from sampling to actuation as well as multi-rate control systems. The logic of the interface automaton is similar to the one discussed above, the only difference being the existence of a single wait state, which is maintained while $t < T$. When $t = T$, the interface automaton generates the control signals needed to produce output, get input, start SPU and restart timer, and goes back to a_1 waiting till the end of the period, and so on, until stopped.



a) with deadline greater than zero b) with deadline equal to zero (no deadline)

Fig. 6. Interface automata of event-driven actors

Fig. 6 shows interface automata for event-driven actors with and without deadline. The automaton of Fig. 6-a is enabled by the arrival of an external start event (e.g. external event, sync message or another message arrival event), whereupon it generates the control signals needed to get input, start SPU and start a timer measuring

deadline. When $t = D$ the automaton checks the *ready* feedback signal and generates the control signal needed to produce output (*ready = 1*) or a deadline violation exception (*ready = 0*).

An event-driven actor may not have a deadline, e.g. a non-terminal actor of a phase-aligned transaction that generates output signals immediately after its computation is finished (see Fig. 6-b). It can be shown that such an actor has a synchronous execution semantics ($D = 0$), whereas the terminal actor has a clocked synchronous semantics ($D = D_{trans}$), i.e. it inherits the end-to-end deadline of the transaction, such that its output signals are generated at the transaction deadline instant [9]. Fig. 7 shows an interface automaton describing the behaviour of a terminal transaction actor. The latter is triggered by a message arrival event but its deadline timer is started by the global start event when the transaction is released (see e.g. Fig. 3).

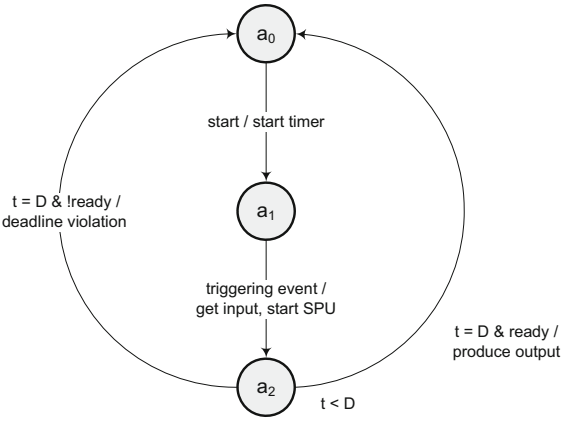


Fig. 7. Interface automaton for a terminal transaction actor

The presented modeling technique has implications for system design, since interface automata can be used as design models for operational components conducting the execution of system actors. It has also implications for system verification, since interface automata are essentially timed automata that can be used as analysis models for verification tools like e.g., Uppaal or simulation environments such as Simulink. In particular, interface automata can be used to develop analysis models that preserve the timed multitasking semantics of the design models. In that case, the actor is modelled by a pair consisting of an interface automaton and a functional automaton (or Simulink subsystem) modelling the behaviour of the SPU. Fig. 8 depicts such a construct using Uppaal notations.

The interface automaton is synchronized with another automaton (e.g. a sync message generator or a preceding actor) via the *start* broadcast channel, whereas the interface and functional automata synchronize with each other by means of the *start SPU* channel and the *ready* feedback signal. The functional automaton has two states - *initial* (a_0) and *executing* (a_1), whereby the transition from a_0 to a_1 is labelled with one or more functions from input to output signals, specifying the signal transformations performed by the SPU. State a_1 is a timed state with an invariant $t < R$, where t is the

clock measuring task execution time and R is the task response time, which can be determined through response-time analysis. It is exited when $t = R$, whereby the feedback signal *finished* is set to *true*.

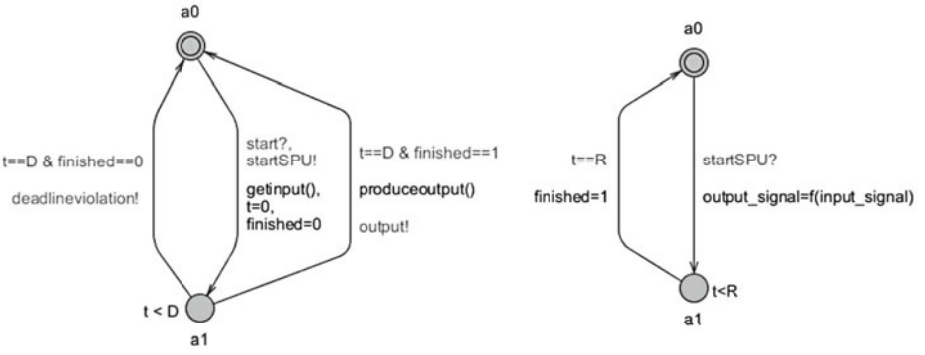


Fig. 8. Uppaal model of event-driven actor with deadline

The timed automata shown in Fig. 8 represent an event-driven actor with deadline, which can be triggered by external events such as global timing events, message arrival events, etc., modelled by the synchronization channel *start*. In a phase-aligned transaction, the corresponding signal will be generated by a predecessor actor via its *output* synchronization channel, which will be activated when the predecessor output is produced. In that case data can be exchanged via a shared data structure.

Similar models can be also developed for the other types of interface automata discussed in this section. Ultimately, the presented technique can be used to transform a system design model into a consistent analysis model accepted by verification tools, such as Uppaal. In particular, each actor is modelled as a pair of interface and functional automata, and actor interaction – as broadcast communication through shared data structures used by the corresponding interface automata, as well as broadcast synchronization channels modelling message arrival events.

4 Implementation Aspects of Interface Automata

The conceptual actor model of Fig. 4 can be used as a design model, whereby the interface automaton is implemented as an application component (state machine) interacting with the input latch, signal-processing unit and output latch. The interface automaton can also be viewed as an operating system component, which processes relevant events in accordance with the corresponding behavioural pattern, and conducts the execution of the actor by invoking the necessary kernel primitives. This interaction is explained below in more detail, in the context of the HARTEXμ kernel [10].

Interface automata can be implemented as service routines that are invoked by the kernel Event Manager while processing the corresponding timing and external events, whereas the signal-processing units are mapped onto actor tasks (task bodies). Likewise, I/O latches are implemented as task interface routines – task I/O, which are executed in separation from the main functions of the corresponding tasks.

In that case the interaction between the interface automaton and the actor task is accomplished by means of the corresponding kernel primitives (see Fig. 9). The primitive *release(task)* is used to implement the control actions *get input* and *start SPU* of the conceptual model. Specifically, it is used to register a task - its input and main body for execution in the corresponding kernel data structures, i.e. Boolean vectors that are used to emulate system queues [10]. Likewise, the primitive *finish(task)* is used to implement the action *produce output* by registering the task output in the corresponding kernel vector.

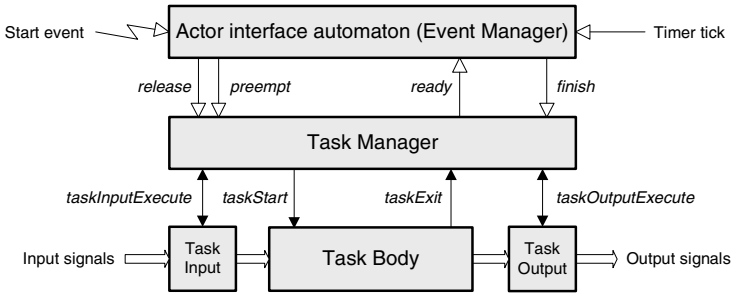


Fig. 9. Interaction between actor interface automaton and actor task

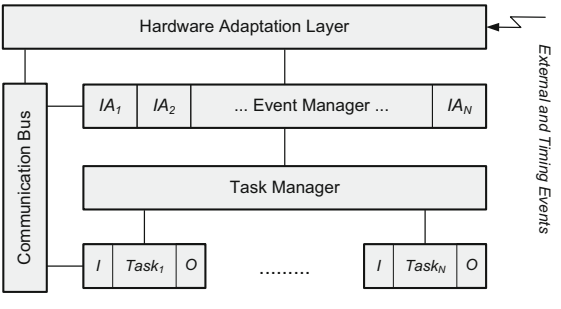


Fig. 10. DTM kernel architecture

The Event Manager may be invoked by interrupt service routines processing external or tick interrupts. When activated, it processes all interface automata that are enabled for execution, which may result in one more newly released tasks having higher priority than the currently running task. Therefore, the execution of the Event Manager ends up with the invocation of the Task Manager primitive *preempt()*, which executes *atomically* all registered task outputs and then task inputs in order to observe the precedence relation between producer and consumer actors exchanging signals at the particular time instant; it proceeds further by selecting the highest-priority task to execute from among all registered tasks and the currently running task, which may be preempted or continued depending on task priorities. A high-priority task will be started immediately, whereas a lower priority task will wait for its turn to execute and will be eventually started when the previously running and/or registered higher priority tasks come to an end. The execution of a running task is finished by returning to

the Task Manager, which resets its registration bit, thereby generating the feedback signal *ready*.

The presented mechanisms constitute a new solution, which allows for very fast and uniform treatment of events and tasks in the context of a real-time kernel consisting of three main subsystems: *Event Manager* incorporating actor interface automata, *Task Manager* scheduling the execution of actor tasks, and a *Communication Bus* used to transparently exchange signals between communicating actor tasks via broadcast event notification and communication primitives (see Fig. 10). More details about the implementation of these and other kernel primitives can be found in [10].

5 Related Research

Distributed Timed Multitasking has been inspired by the original *Timed Multitasking* model [4] and is similar to the logical execution time (LET) model adopted in the *xGiotto* language [6]. However, both of these models employ port-based communication between actors, whereas DTM employs broadcast communication using labeled state messages (signals). This is a flexible solution, which rules out artifacts such as ports, mailboxes, operational interfaces with call-return semantics, etc., thus providing for reconfigurable and truly open distributed embedded systems.

The adopted communication mechanism supports transparent communication and is characterized by complete separation of computation and communication, since signal drivers are executed in separation from actor tasks and from each other. That is not the case with port-based objects, where ports are usually defined as communication objects whose methods are invoked within task I/O drivers in a conventional call-return fashion, much in the same way as operational interfaces, see e.g. [4]. Consequently, the communication pattern is 'hardwired' in the code of I/O drivers and cannot be reconfigured without reprogramming. Furthermore, in that implementation ports are conceived as shared data structures, which are not suitable for distributed applications.

DTM has certain similarities with the models of computation used in synchronous languages [1]. At the same time, there are some notable differences, and in particular:

- True actor-level concurrency vs. conceptual concurrency, which is 'compiled away' during the translation of synchronous programs
- Constant *non-zero* reaction time vs. instantaneous (zero-time) reaction assumed by perfectly synchronous systems.

The last feature facilitates the engineering of distributed systems and eliminates major problems related to fixpoints, instantaneous loops, etc. Furthermore, the synchronous model does not address the problem of I/O jitter because of the very nature of the synchrony hypothesis, whereas it is practically eliminated with DTM due to the constant delay from sampling to actuation inherent to that model.

6 Conclusion

The paper presents an operational specification of Distributed Timed Multitasking in terms of actor interface automata. This modeling technique has implications for system design, since interface automata can be used as design models implemented as

application or operating system components. It has also implications for system verification, since interface automata are essentially timed automata that can be easily transformed into analysis models used in model-checking tools like Uppaal or simulation environments such as Simulink.

Interface automata can be used to efficiently implement the event management subsystem of a DTM kernel, which allows for fast and uniform treatment of events and tasks in an operational environment consisting of three main subsystems: *Event Manager* incorporating actor interface automata, *Task Manager* scheduling the execution of actor tasks, and a *Communication Bus* used to transparently exchange labeled messages (signals) between communicating actor tasks.

This architecture has been used to implement the latest version of the HARTEXμ kernel. Research is now going on, aimed at a hardware implementation of the Event Manager and eventually – the entire kernel, with application tasks running in the soft cores of a multi-core FPGA chip.

References

1. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., de Simone, R.: The Synchronous Languages 12 Years Later. Proc. of the IEEE 91(1), 64–83 (2003)
2. Marti, P., Villa, R., Fuertes, J.M., Fohler, G.: On Real-Time Control Task Schedulability. In: Proceedings of the European Control Conference, Porto, Portugal, pp. 2227–2232 (September 2001)
3. Jantsch, A.: Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation. Morgan Kaufmann, San Francisco (2003)
4. Liu, J., Lee, E.A.: Timed Multitasking for Real-Time Embedded Software. IEEE Control Systems Magazine: Advances in Software Enabled Control, 65–75 (February 2003)
5. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: GIOTTO: a Time-Triggered Language for Embedded Programming. Proc. of the IEEE 91, 84–99 (2003)
6. Ghosal, A., Henzinger, T.A., Kirsch, C.M., Sanvido, M.A.: Event-Driven Programming with Logical Execution Times. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 357–371. Springer, Heidelberg (2004)
7. de Alfaro, L., Henzinger, T.A.: Interface Automata. In: Proc of the 8th European Software Engineering Conference ESEC 2001, Austria (2001)
8. Angelov, C., Ke, X., Sierszecki, K.: A Component-Based Framework for Distributed Control Systems. In: Proc. of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications SEAA 2006, Cavtat, Dubrovnik, Croatia, pp. 20–27 (August-September 2006)
9. Angelov, C., Sierszecki, K., Guo, Y.: Formal Design Models for Distributed Embedded Control Systems. In: Proc. of the 2nd International Workshop on Model Based Architecting and Construction of Embedded Systems ACES-MB 2009, Denver, Colorado, USA, pp. 43–57 (October 2009)
10. Sierszecki, K., Angelov, C., Ke, X.: A Run-Time Environment Supporting Real-Time Execution of Embedded Control Applications. In: Proc. of the 14th International IEEE Conference on Embedded and Real-Time Computing Systems and Applications RTCSA 2008, Kaohsiung, Taiwan (August 2008)

Building a Time- and Space-Partitioned Architecture for the Next Generation of Space Vehicle Avionics*

José Rufino, João Craveiro, and Paulo Verissimo

University of Lisbon, Faculty of Sciences, LaSIGE
ruf@di.fc.ul.pt, jcraveiro@lasige.di.fc.ul.pt, pjv@di.fc.ul.pt

Abstract. Future space systems require innovative computing system architectures, on account of their size, weight, power consumption, cost, safety and maintainability requisites. The AIR (ARINC 653 in Space Real-Time Operating System) architecture answers the interest of the space industry, especially the European Space Agency, in transitioning to the flexible and safe approach of having onboard functions of different criticalities share hardware resources, while being functionally separated in logical containers (partitions). Partitions are separated in the time and space domains. In this paper we present the evolution of the AIR architecture, from its initial ideas to the current state of the art. We describe the research we are currently performing on AIR, which aims to obtain an industrial-grade product for future space systems, and lay the foundations for further work.

1 Introduction

Space systems of the future demand for innovative embedded computing system architectures, meeting requirements of different natures. These systems must obey to strict dependability and real-time requirements. Reduced size, weight and power consumption (SWaP), along with low wiring complexity, are also crucial requirements both for safety reasons and to decrease the overall cost of a mission. A modular approach to software enabling component reuse among the different space missions also benefits the cost factor. At the same time, such an approach allows independent validation and verification of components, thus easing the software certification process.

A typical spacecraft onboard computer has to host a set of avionics functions, such as the Attitude and Orbit Control Subsystem (AOCS), the Telemetry,

* This work was partially developed within the scope of the European Space Agency Innovation Triangle Initiative program, through ESTEC Contract 21217/07/NL/CB, Project AIR-II (ARINC 653 in Space RTOS — Industrial Initiative, http://air.di.fc.ul.pt). This work was partially supported by Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology), through the Multiannual Funding and CMU-Portugal Programs and the Individual Doctoral Grant SFRH/BD/60193/2009.

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 179–190, 2010.

Tracking, and Command (TTC) subsystem, and one or more payload subsystems [13]. The traditional approach to space computing systems was to grant dedicated hardware resources to each of these functions.

However, the ongoing trend goes towards the integration of multiple functions, so that they share the same hardware resources. While satisfying SWaP requirements, this introduces potential safety risks, since the applications supporting those functions may have different degrees of criticality and predictability, and/or may originate from multiple sources. The architectural principle proposed to cope with function integration has onboard applications being separated in logical containers, called partitions. Partitioning allows achieving both fault containment and independent software verification/validation capabilities. The safety of this approach is enforced through robust temporal and spatial partitioning (TSP) [25, 28]. Temporal partitioning concerns partitions not interfering with the fulfilment of each other's real-time requisites, while spatial partitioning encompasses the usage of separate addressing spaces dedicated to each partition. The aeronautic industry went through a similar process [32], introducing the Integrated Modular Avionics (IMA) [1] and ARINC 653 [2,3] specifications.

In this paper, we present our past, present, and future research work on the AIR (ARINC 653 in Space RTOS) architecture for TSP aerospace systems. AIR has been prompted by the interest of the space industry in the adoption of TSP concepts [25,30], especially the European Space Agency (ESA), which is currently active in this matter within the TSP Working Group [33]. The National Aeronautics and Space Administration (NASA) has expressed a similar interest for its next generation of space exploration vehicles [14,12].

The design of the AIR architecture incorporates state-of-the-art features, such as coexistence of real-time operating systems (RTOS) and generic non-real-time ones, advanced timeliness control and adaptation mechanisms, and flexible development and integration tools. It also foresees extensions to the architecture so as to take advantage of multicore platforms.

The remainder of this paper is organized as follows. Section 2 details the successive evolution steps of the AIR architecture, from its inception ideas to the current state of the art. Section 3 details further the AIR architecture. Section 4 describes the work currently being done on the AIR architecture. Section 5 presents open research issues which we plan to tackle. Finally, Section 6 closes the paper with some concluding remarks.

2 Evolution of AIR Design Solutions

The first steps of the AIR project were performed under commissioning of ESA, within the scope of the Innovation Triangle Initiative program.

2.1 ARINC 653 Interface in RTEMS

The initial idea was to perform a feasibility study to adapt the Real-Time Executive for Multiprocessor Systems (RTEMS) [18] to offer the application interface

and functionality required by the ARINC 653 specification [2,9]. The proposed solution involved an analysis of the RTEMS modules needing to be modified, extended, removed or added to cope with the missing ARINC 653 functionality. This approach was never strictly followed [23].

2.2 Single-Executive Core (SEC)

A more interesting solution, devised on the early stages of development, was to make the architecture design independent from the underlying RTOS kernel.

Stemming directly from the ARINC 653 specification, the original AIR design approach integrates a standard Application Executive (APEX) interface module, which maps into the service interface of a single RTOS kernel. The architecture of this Single-Executive Core (SEC) design, illustrated in Fig. 1, includes the RTOS kernel (providing functions such as process management, time and clock management, and interprocess synchronization and communication), and a module integrating the system-specific functions associated to the underlying processor infrastructure and to the specific platform hardware resources.

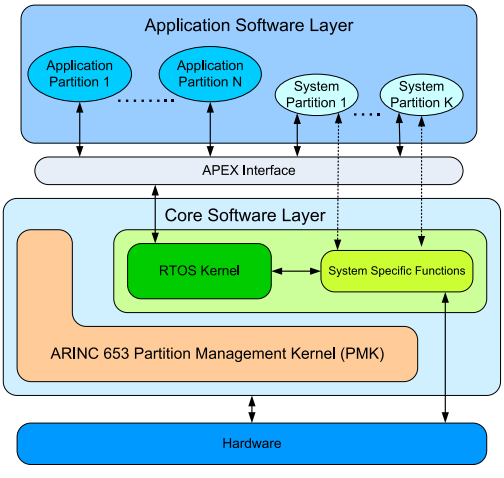


Fig. 1. Single-executive core design approach

The core functionality needed by the ARINC 653 specification (cyclic partition scheduling and priority-based process scheduling) was implemented by a specific module, the *ARINC 653 Partition Management Kernel*, shown in Fig. 1.

The SEC design exhibits an optimal memory footprint size, since there is only one instance of each component. Given that access to the APEX interface and RTOS kernel is shared among all partitions, the integrity and fault confinement attributes are restricted to the application software layer (see Fig. 1).

The SEC design and its proof-of-concept prototype have been very helpful to understand the realm of time and space partitioning and have proved the feasibility of implementing the ARINC 653 functionality making use of off-the-shelf

operating systems. The proof-of-concept prototype was developed and demonstrated using RTEMS 4.6.6 on an Intel IA-32 platform.

2.3 Multi-Executive Core (MEC)

To improve the integrity and fault confinement properties of the AIR architecture, a Multi-Executive Core (MEC) design was approached. In MEC, the separation in logical containers is extended throughout all layers, as shown in Fig. 2. This is achieved by providing an APEX interface, an RTOS kernel and possibly system specific functions on a per-partition basis. From the SEC approach, the MEC design solution preserves the independence from a given operating system (OS) and permits a homogeneous integration of different instances of the same RTOS kernel or a hybrid approach, possibly integrating different RTOS kernels in different partitions, the *Partition Operating Systems* (POS).

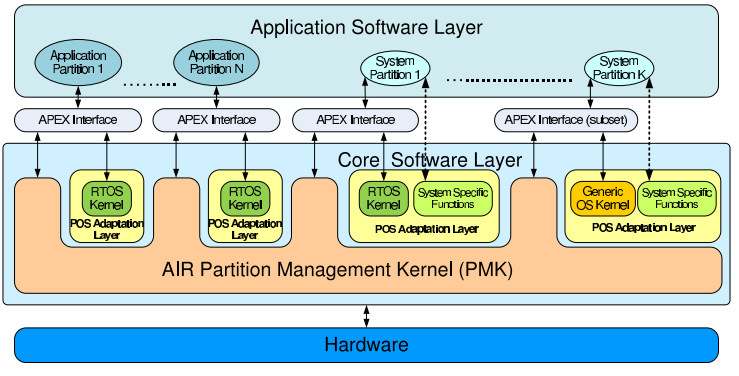


Fig. 2. Multi-executive core AIR architecture

The new *AIR Partition Management Kernel* (PMK) provides the core functionality needed for conformity with the ARINC 653 specification [2], separating the partition and process scheduling functions. Process scheduling is ensured by the native RTOS process scheduler of each partition. The AIR PMK also includes partition management and support for interpartition communication.

A proof-of-concept prototype for the MEC architecture was developed and demonstrated using RTEMS 4.8 as partition operating system. Two processor platforms have been approached: an Intel IA-32 platform, and SPARC ERC32 and SPARC LEON-based platforms.

2.4 Comparison between SEC and MEC Design Solutions

The MEC design solution mostly improves on a set of relevant attributes, in comparison with the SEC solution. A system-wide suboptimal footprint size is outweighed by the benefits obtained in terms of flexibility, configurability,

integrity and fault confinement. The MEC design is further flexible in the sense that it allows the integration of different RTOS kernel instances in different partitions. A comparison detailing these improvements is presented in Table 1.

Table 1. Comparison between the AIR design solutions' attributes

	Single-executive core (SEC)	**Multi-executive core (MEC)**	
RTOS Integration	shared	per partition	
Flexibility	fair	good	
Configurability	system-wide	per partition	
Integrity and Fault Confinement	application layer	all layers	
Footprint Size	optimal (system-wide)	optimal (per partition)	
Development Tools	canonical	canonical	canonical and specific
Bootstrap Method	single image	single image	multiple images

The remaining design attributes in Table 1 do address the requirements of the development tools and of the application bootstrap methodology. One limitation of the application production toolchain concerns a common inability of canonical link editors to combine in a single object the different instances of the same RTOS kernel, since they use the same naming references. A methodology is needed to tackle this problem. A tag filter utility, which appends a partition identifier to each public symbol, is used. The objects of each partition may afterwards be combined by a canonical link editor into a single linked object. The MEC design solution, allowing multiple objects to be specified for bootstrapping, opens room for the dynamic update of individual partition applications.

3 AIR System Architecture

Figure 2 also shows some later additions to the AIR architecture definition, which we will now discuss.

3.1 Temporal and Spatial Partitioning

The robust partitioning approach defined in the AIR architecture implies the temporal and spatial separation of the different operating systems and its applications in integrity and criticality containers, defined by partitions. Temporal partitioning is achieved through a two-level hierarchical scheduling scheme pictured in Fig. 3. In the first level, partitions are scheduled according to a cyclic sequence of fixed time slices. Inside each partition, processes compete with each other according to the native process scheduler of the partition; in the case of RTOSs, this is typically a dynamic priority-based scheduler [24].

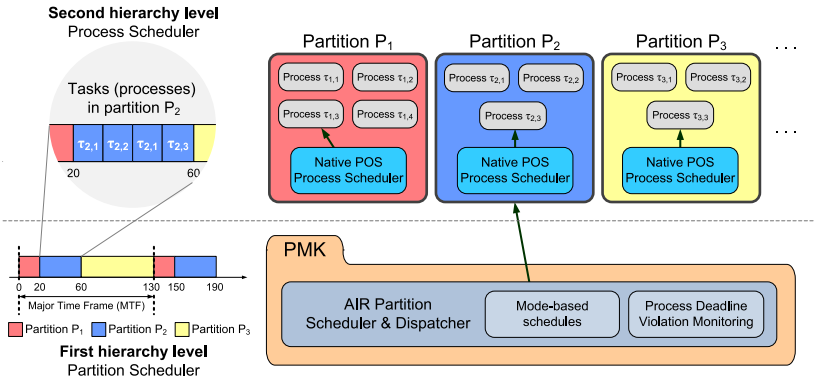


Fig. 3. AIR two-level hierarchical scheduling

Partitions spatially encapsulate the addressing spaces of the contained POS and applications. No component of a given partition can directly access the addressing space of other partitions, thus guaranteeing that partitions do not interfere with each other [22,24]. A highly modular design approach in the support of AIR spatial partitioning was followed, with requirements (specified in configuration files with the assistance of development tools support) being described in runtime through a high-level processor independent abstraction layer [22]. Hardware-mapping descriptors are provided per partition, primarily corresponding to the several levels of execution of an activity (e. g., application, POS kernel and AIR PMK) and to its different memory blocks (e. g., code, data and stack).

3.2 Advanced Timeliness Control and Adaptation Mechanisms

A basic partition scheduling scheme, with a single partition scheduling table defined at integration time, is very limiting regarding the different temporal characteristics a space mission can adopt in distinct phases of its operation (e. g., takeoff, flight, exploration) or regarding the accommodation of component failures. The AIR advanced design addresses this issue by introducing support for multiple *mode-based partition schedules*. The basic partition scheduling scheme is extended to allow multiple schedules to be defined. At execution time, authorized partitions may request switching between the different partition schedules [5,24].

Another timeliness control mechanism introduced in AIR is *process deadline violation monitoring*. During the execution of the system, it may be the case that a process exceeds its deadline; this can be caused by a malfunction or because that process's worst-case execution time (WCET) was underestimated at system configuration and integration time. Other factors related to faulty system planning (such as violation of the partitions' timing requirements) can be predicted and avoided using offline tools [7], addressed in Sect. 4.1. The process deadline violation monitoring procedure is optimized regarding deadline violation detection latency and regarding computation complexity so as not to have minimal temporal interference with the rest of the system, since it is performed inside the

system clock interrupt service routine: the earliest deadline is checked; following deadlines may subsequently be verified until one has not been missed [5,24].

3.3 Flexible Partition Operating System Integration

The *AIR POS Adaptation Layer* (PAL) encapsulates each POS providing a common interface to the surrounding components (AIR PMK, APEX). This way, the necessary changes to support using a new family or version of POS are circumscribed to a smaller component, and previous or ongoing verification, validation and/or certification efforts on more complex ones are not hindered [5,6].

3.4 Integration of Generic Operating Systems

The foreseen heterogeneity between POSs is also being extended to include generic non-real-time OSs, such as Linux, answering to a recent trend in the aerospace industry. This is motivated by the lack of relevant functions in most RTOSs, which are commonly provided by generic non-real-time operating systems. Porting these functions (e. g., scripting language interpreters) to RTOSs can be a complicated and error-prone task [16]. An embedded variant of Linux has been approached, and wields a fully functional OS with a minimal size compatible with typical space missions requirements [8,5].

To ensure that a non-real-time OS cannot undermine the overall time guarantees of the system by disabling or diverting system clock interrupts, instructions that could allow this are wrapped by low-level handlers (paravirtualized) [5,6].

3.5 Flexible Portable APEX

The ARINC 653 specification defines a standard interface between applications and the core software layer [2], the APEX interface. The *AIR APEX interface* component (Fig. 2) supports this feature, exploiting the availability of AIR PAL-related functions and implementing the advanced notion of *Portable APEX* [26].

3.6 AIR Health Monitoring (HM)

The AIR Health Monitor is responsible for handling hardware and software errors (like deadlines missed, memory protection violations, or hardware failures). The aim is to isolate errors within its domain of occurrence: process level errors will cause an application error handler to be invoked, while partition level errors trigger a response action defined in a system configuration table. Errors detected at system level may lead the entire system to be stopped or reinitialized [22].

3.7 Interpartition Communication

Interpartition communication aims to support the transfer of information between partitions, and its relation with spatial partitioning implies the use of specific executive interface services encapsulating and providing the transfer of data

from one partition to another without violating spatial segregation constraints. The core of AIR interpartition communication mechanisms are integrated at the Portable APEX interface level. Memory protection and, if required, memory-to-memory copy mechanisms are managed at the AIR PMK level [22,24].

The interpartition communication abstractions required for conformity with ARINC 653, sampling ports and queuing ports, model each partition's way to communicate (send *or* receive messages) through a communication channel.

4 Present Lines of Work

4.1 Scheduling and Composability

In [7], we discuss how we can profit from composability properties inherent to the build and integration process of AIR-based systems to allow validating scheduling requirements independently at different levels of the build and integration process. We proposed the development of rules, techniques and tools to support this purpose, so as to provide both schedulability analysis capabilities but also tool-assisted generation of partition scheduling tables.

Schedulability results for TSP systems allow extensions to the software build and integration processes of AIR-based systems, for the benefit of both application developers (Fig. 4(a)) and system integrators (Fig. 4(b)). This benefit obviously depends on software tools, using such results, being made available to them. The goal of application developers having a scheduling analysis phase introduced in their production cycle is for them to be able to independently analyse the feasibility of their applications, provided the timing requirements (period, WCET, deadline, etc.) of the composing processes. The information of these timing requirements can be either estimated by the developers or tentatively determined through static code analysis [19].

4.2 Multicore

Multicore processors are paving their way into the realm of embedded systems [17], but their use in TSP platforms has not been addressed in detail [5]. In this sense, we pursue the extension of TSP concepts to allow multicore-enabled AIR-based systems, which feasibility shall be backed up by both schedulability and safety considerations. The applicability of multicore includes strengthening overall safety through adaptive fault tolerance mechanisms, and augmenting the integration potential of a single system with the contribute of different facets of parallelism. The tools proposed in [7] should in this case be extended to accommodate multicore support.

This research line includes the profound analysis of the impact of parallelism, both intrapartition parallelism (i. e., between processes) and between partitions. The approach to intrapartition parallelism aims to understand the advantages and drawbacks in distributing processes in a partition among processor cores. Concerning parallelism between partitions, two scheduling approaches will be

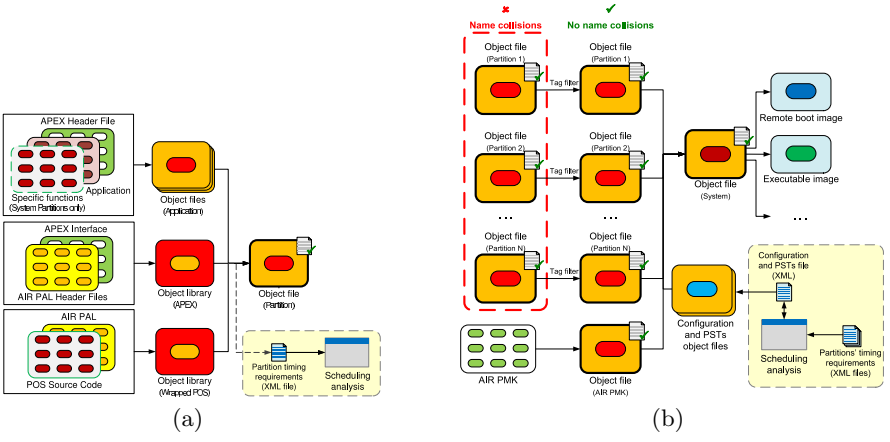


Fig. 4. Introduction of scheduling analysis features for (a) application developers, and (b) system integrators

studied: *(i)* static (extending system configuration mechanisms, to allow explicit definition of when and how parallelism between partitions occurs), and; *(ii)* semi-dynamic (extending configuration mechanisms, to allow expressing restrictions and dependencies that will guide the activity of a dynamic partition scheduler with support for parallelism between partitions) [5].

4.3 Remote and Online Application Update

The Mars Rover Pathfinder is an example of a mission where the (in this case fortuitous) possibility to modify the mission's configuration remotely was crucial for its survival [15]. Due to the impossibility of direct access to the spacecraft, it is extremely important to have the possibility for the onboard system to remotely receive software updates [20].

At this stage, AIR abstracts from issues inherent to the communication between the spacecraft and the ground station, focusing on the management and treatment of the update information (integrity, correctness, domain of application). Remote software update may have to cope with critical software components that must be updated without interruptions to their execution.

Other interesting issues include remote system monitoring and modification of system-wide control parameters (such as partition scheduling tables). These will increase the extent to which the advanced timeliness control and adaptation mechanisms of AIR are taken advantage of.

5 Future

5.1 Operating System Integration

The work mentioned in Sect. 3.4 shall be extended in two ways. On the one hand, the study involving embedded Linux [8,5] will evolve into a fully functional

Linux integration. On the other hand, the principles should be applied for the integration of other generic non-real-time operating system, such as Windows through the Windows Research Kernel [27]. Furthermore, for specific application support, the integration of other RTOS kernels, such as eCos, is also envisaged.

5.2 Sensors, Actuators and Networks

Any spacecraft needs interfacing with surrounding environment, through sensors and actuators. For instance, the AOCS function needs to get information from star/Sun sensors and reference gyroscopes. On the other hand, AOCS needs to actuate on reaction wheels and propulsion drive thrusters. The safety of these interactions with the environment should be supported by extending the spatial partitioning mechanisms to input/output (I/O) addressing spaces.

A particular case of I/O functions is network communication. This may include wired network interfaces, such as: legacy MIL-STD-1553 [10] systems; *dependable* Controller Area Network (CAN) buses [21]; high-rate SpaceWire [11] and TTEthernet [31] links. For some space systems, such as planetary robotic explorers, wireless sensor networks with improved dependability and timeliness may be of the utmost importance for sensing and coordination actions [29].

5.3 Information Security

The AIR architecture still requires the incorporation of security concerns. One option for partitioned security is the notion of Multiple Independent Levels of Security and Safety (MILS) [4]. This implies that, at the application level, execution is confined to the application's partition, with controlled communication with the remaining partitions. All communication passes through the security components, which can include monitoring and cryptographic mechanisms.

To fulfil MILS requisites, the AIR architecture will incorporate the provision of privacy- and authenticity-capable interpartition communication services, using the cryptographic mechanisms and algorithms most adequate to the characteristics of TSP systems (for encryption and, possibly, key distribution). This extension includes the analysis of the architectural, hardware and cryptographic algorithm requirements for this functionality.

6 Conclusion

This paper presents the evolution of the AIR architecture, from its initial ideas to the current state of the art. AIR targets space systems of the future, and current work aims to turn it into an industrial-grade product. The first approach to AIR was a single-executive core design, but soon evolved in to a multi-executive one. Subsequent research work provided AIR with flexible support to different partition operating systems (both real-time and non-real-time) and advanced timeliness control and adaptation mechanisms. Current work focuses on schedulability

issues, taking advantage of multicore platforms, and remote online update of applications. Research directions for the future include expansion of support to new operating systems, interfaces with input/output devices (including networking), and security (privacy and authenticity) of information exchanges.

Acknowledgment

The authors would like to thank all the researchers, engineers and collaborators who worked throughout the time in the AIR Technology (AIR and AIR-II projects), at FCUL, GMV Portugal, Thales Alenia Space, and ESA–ESTEC.

References

1. AEEC: Design guidance for Integrated Modular Avionics. ARINC Report 651-1 (November 1997)
2. AEEC: Avionics application software standard interface, part 1 - required services. ARINC Specification 653P1-2 (March 2006)
3. AEEC: Avionics application software standard interface, part 2 - extended services. ARINC Specification 653P2-1 (December 2008)
4. Alves-Foss, J., Harrison, W.S., Oman, P., Taylor, C.: The MILS architecture for high-assurance embedded systems. Int. J. of Embedded Systems 2, 239–247 (2006)
5. Craveiro, J.: Integration of generic operating systems in partitioned architectures. M.Sc. thesis, Faculty of Sciences, University of Lisbon, Lisbon, Portugal (2009)
6. Craveiro, J., Rufino, J., Schoofs, T., Windsor, J.: Flexible operating system integration in partitioned aerospace systems. In: Actas do INForum - Simpósio de Informática 2009, Lisbon, Portugal, pp. 49–60 (September 2009)
7. Craveiro, J., Rufino, J.: Schedulability analysis in partitioned systems for aerospace avionics. In: Proc. 15th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA 2010), Bilbao, Spain (September 2010)
8. Craveiro, J., Rufino, J., Almeida, C., Covelo, R., Venda, P.: Embedded Linux in a partitioned architecture for aerospace applications. In: Proc. 7th ACS/IEEE Int. Conf. on Computer Systems and Applications (AICCSA 2009), Rabat, Morocco, pp. 132–138 (May 2009)
9. Diniz, N., Rufino, J.: ARINC 653 in space. In: Proc. DASIA 2005 "Data Systems In Aerospace" Conf., Edinburgh, Scotland (June 2005)
10. ECSS: Space engineering: Interface and communication protocol for MIL-STD-1553B data bus onboard spacecraft. Standard ECSS-E-50-13 Draft C, ESA Requirements and Standards Division (May 2008)
11. ECSS: Space engineering: SpaceWire — links, nodes, routers and networks. Standard ECSS-E-ST-50-12C, ESA Requirements and Standards Division (July 2008)
12. Fletcher, M.: Progression of an open architecture: from Orion to Altair and LSS. Tech. rep., Honeywell International (May 2009)
13. Fortescue, P.W., Stark, J.P.W., Swinerd, G. (eds.): Spacecraft Systems Engineering, 3rd edn. Wiley, Chichester (2003)
14. Hodson, R., Ng, T.: Avionics for exploration. In: NASA Technology Exchange Conference, Galveston, TX, USA (November 2007)
15. Jones, M.: What really happened on Mars Rover Pathfinder. The RISKS Digest - Forum on Risks to the Public in Computers and Related Systems 19(49) (December 1997), http://catless.ncl.ac.uk/Risks/19.49.html

16. Kinnan, L.: Application migration from Linux prototype to deployable IMA platform using ARINC 653 and Open GL. In: Proc. 26th IEEE/AIAA Digital Avionics Systems Conference, Dallas, TX, USA, pp. 6.C.2-1–6.C.2-5 (October 2007)
17. Mignolet, J.Y., Wuyts, R.: Embedded multiprocessor systems-on-chip programming. IEEE Software 26(3), 34–41 (2009)
18. OAR - On-Line Applications Research Corporation: RTEMS C Users Guide, 4.8 (February 2008)
19. Pushner, P., Koza, C.: Calculating the maximum execution time of real-time programs. Journal of Real-Time Systems 1, 160–176 (1989)
20. Rosa, J., Craveiro, J., Rufino, J.: Exploiting AIR composability towards spacecraft onboard software update. In: Actas do INForum - Simpósio de Informática 2010, Braga, Portugal (September 2010)
21. Rufino, J., Almeida, C., Verissimo, P., Arroz, G.: Enforcing dependability and timeliness in Controller Area Networks. In: Proc. 32nd Ann. Conf. of the IEEE Industrial Electronics Society (IECON'06), Paris, France (November 2006)
22. Rufino, J., Craveiro, J., Schoofs, T., Tatibana, C., Windsor, J.: AIR Technology: a step towards ARINC 653 in space. In: Proc. DASIA 2009 "Data System In Aerospace" Conf., Istanbul, Turkey (May 2009)
23. Rufino, J., Filipe, S., Coutinho, M., Santos, S., Windsor, J.: ARINC 653 interface in RTEMS. In: Proc. DASIA 2007 "DAta Systems In Aerospace" Conf., Naples, Italy (June 2007)
24. Rufino, J., Craveiro, J., Verissimo, P.: Architecting robustness and timeliness in a new generation of aerospace systems. In: Casimiro, A., de Lemos, R., Gacek, C. (eds.) Architecting Dependable Systems. LNCS, vol. 7. Springer, Heidelberg (2010)
25. Rushby, J.: Partitioning in avionics architectures: Requirements, mechanisms and assurance. NASA Contractor Report CR-1999-209347, SRI International, California, USA (Jun 1999)
26. Santos, S., Rufino, J., Schoofs, T., Tatibana, C., Windsor, J.: A portable ARINC 653 standard interface. In: Proc. IEEE/AIAA 27th Digital Avionics Systems Conf. (DASC '08), St. Paul, MN, USA (October 2008)
27. Schöbel, M., Polze, A.: Kernel-mode scheduling server for CPU partitioning: a case study using the Windows Research Kernel. In: Proc. 2008 ACM Symp. on Applied Computing (SAC 2008), pp. 1700–1704. ACM, Fortaleza (2008)
28. Seyer, R., Siemers, C., Falsett, R., Ecker, K., Richter, H.: Robust partitioning for reliable real-time systems. In: Proc. 18th Int. Parallel and Distributed Processing Symp., pp. 117–122 (April 2004)
29. Souza, J.L.R., Rufino, J.: Characterization of inaccessibility in wireless networks: a case study on IEEE 802.15.4 standard. In: Proc. IESS International Embedded Systems Symposium '09, Langenargen, Germany (September 2009)
30. Terraillon, J.L., Hjortnaes, K.: Technical note on on-board software. European Space Technology Harmonisation, Technical Dossier on Mapping, TOSE-2-DOS-1, ESA (February 2003)
31. TTTech.: TTEthernet specification. Document D-INT-S-10-002, TTTech. Computertechnik AG (November 2008)
32. Watkins, C., Walter, R.: Transitioning from federated avionics architectures to Integrated Modular Avionics. In: Proc. 26th IEEE/AIAA Digital Avionics Systems Conf. (DASC 2007), Dallas, TX, USA (October 2007)
33. Windsor, J., Hjortnaes, K.: Time and space partitioning in spacecraft avionics. In: Proc. 3rd IEEE Int. Conf. on Space Mission Challenges for Information Technology (SMC-IT 2009), Pasadena, CA, USA, pp. 13–20 (July 2009)

EMWF: A Middleware for Flexible Automation and Assistive Devices

Ting-Shuo Chou[2], Yu Chi Huang[2], Yung Chun Wang[1], Wai-Chi Chen[3], Chi-Sheng Shih[3], and Jane W.S. Liu[1]

[1] Institute of Information Science, Academia Sinica, Taipei, Taiwan
{wych,janeliu}@iis.sinica.edu.tw
[2] Computer Science Department, National Tsing-Hua University, Taiwan
[3] Computer Science and Information Engineering Department, National Taiwan University, Taipei, Taiwan

Abstract. EMWF (Embedded Workflow Framework) is an open source middleware for flexible (i.e., configurable, customizable and adaptable), user-centric automation and assistive devices and systems. EMWF 1.0 provides a light-weight workflow manager and engines on Windows CE, Windows XP Embedded, and Linux. It is for small embedded automation devices. EMWF 2.0 also provides basic message passing and real-time scheduling mechanisms and workflow communication facility. This paper describes EMWF 1.0 and extensions in EMWF 2.0, as well as case studies on workflow-based design and implementation as motivations for EMWF and the extensions.

Keywords: Workflow-based architecture, embedded automation advices, workflow management and engine.

1 Introduction

This paper describes a middleware designed to easy the effort in building high-quality, low-cost personal and home automation and assistive devices and systems. We refer to these devices (and systems) collectively as UCAADS (User-Centric Automation and Assistive Device and Systems). Smart medication dispensers, autonomous appliances, service robots and robotic helpers described in [1-7] are examples. These devices aim to improve the quality of life and self-reliance of their users, including elderly or functionally limited individuals. Other examples of UCAADS are automation tools (e.g., [8, 10]) for use in care-providing institutions for enhancing the quality and reducing the costs of medical and health care.

Despite vast differences in their purposes and functions, UCAADS have many common requirements. First and foremost is *flexibility*; by that, we mean configurability, customizability and adaptability. A flexible device can be configured to support different processes and rely on different support infrastructures. It can be easily customized to suit different users. The device should also be able to adapt to serve the user well over time as the user's needs change.

We have adopted the workflow paradigm [11, 12] as a means to achieve flexibility. This paradigm has been widely used in enterprise systems for automation of business

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 191–203, 2010.

processes. Using this paradigm, the developer decomposes work to be done by the application into basic building blocks called *activities.* An activity may be done by executing a software procedure. We call such an activity a *software activity*. An activity may also be an operation by a hardware device, a message delivery by a network, and so on. Some activities are actions of human users. We call all of these activities *external activities*. Activities are composed into module-level components called *workflows*. The order and conditions under which activities in a workflow are executed and the resources needed for their execution are defined by the developer of the workflow. The definition can be in terms of some programming language (e.g., C# in [11]), a process definition language (e.g., XPDL, WfMC standard XML Process Definition Language [12, 13]), or an execution language (e.g., BPEL, Business Process Execution Language [14]). Workflows can also be defined graphically [11, 15]: In a *workflow graph*, nodes represent activities in (or states of) workflows and directed edges represent transitions between activities (or states).

A key component of a platform for workflow-based applications is the *workflow engine* (or *engine* for short). The engine sequences activities in each workflow, coordinate activities that share resources, schedules and manages activities that are ready to run and in case of software activities, execute them. The engine also provides the applications served by it with *built-in activities*: They are activities that start and stop workflows, and alter the timing and flow paths within workflows during executions. To design and implement a workflow-based application, the developer only needs to define the workflows in the application and provide the resources required by the activities in them and then leaves the engine to integrate workflow components dynamically at runtime as specified.

Today's matured engines and tools for defining, building and executing workflows (e.g., [11-18]) are primarily for applications that automate business processes on enterprise computers and mobile devices. They are not well suited for UCAADS that have embedded components running at high rates and interacting closely with hardware devices. EMWF (Embedded Workflow Framework) presented here is a workflow management system designed specifically for such devices. Typical UCAADS are not as severely power and size constrained as cell phones and PDA's. Consequently, energy consumption and memory footprint requirements of EMWF engine and workflow applications are not as stringent as the requirements of engines and applications for mobile web-based workflow applications (e.g., [17, 18]).

Following this introduction, Section 2 provides illustrative examples to further elaborate workflow-based design and rationales behind EMWF. Section 3 provides an overview of EMWF version 1.0 [19]. Section 4 describes extensions designed to provide EMWF 2.0 (i.e., the next version of EMWF) with communication and real-time capabilities. Section 5 summarizes the paper.

2 Motivations and Rationales

EMWF is written in C. It provides a workflow manager and engines on Linux and Microsoft Windows CE and XP Embedded. We focus here on the relatively mature Windows versions. Hereafter, by EMWF, we mean these versions and call them EMWF 1.0. A description of the Linux version can be found in [19, 20].

2.1 Simple Workflow-Based Devices

Specifically, EMWF 1.0 [19] is for relatively simple embedded devices and system components that run on a processor. An example is an automatic vacuum cleaner. Its workflow-based structure is shown in Fig. 1(a). Most parts of the devices are built from workflows. We omit drivers and components that are hardwired. In the figure, the rectangular boxes represent activities. The middle dotted box encircles software activities executed by the workflow engine on a CPU. External activities are in dotted boxes labeled environment interaction and robot components. These activities are carried out by sensor devices, microcontroller and mechanical parts of the device.

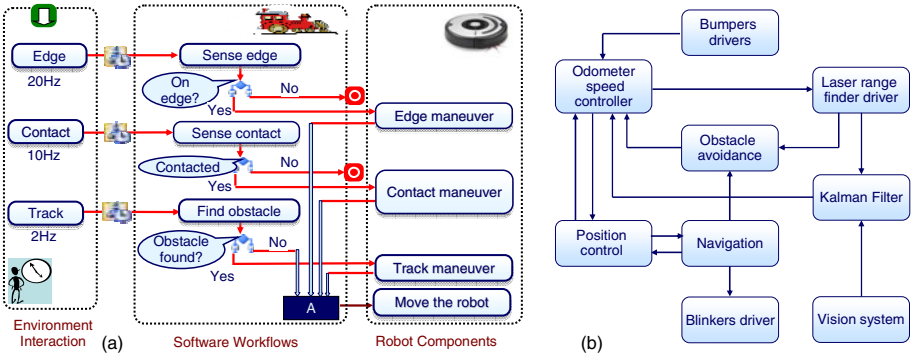


Fig. 1. Parts (a) Example of workflow-based device and (b) tasks in a mobile robot [21]

Table 1. Built-in activities

Generic built-ins	Route	If else (2-way XOR split)	
	Split	Merge	
	Throw	Exception	
	Invoke workflow	Execute workflow	
	Delay /timeout	Set events / timers	
	Wait for events / timers / workflow triggers		
	Start	Stop	While
Built-ins for BC	Superposition	Arbiter	Voter
	Mode change	Push data	Pull data

Table 1 lists built-in activities provided by EMWF. The symbols used to represent some of them are from Microsoft Windows Workflow foundation (WF) GUI editor [11]. Top five rows lists examples of generic built-ins, including start, stop, if else, and wait-for built-ins that are used in the device shown in Fig. 1(a). All applications need some generic built-ins. The bottom row of Table 1 lists built-ins for robotic behavior coordination (BC), including arbiter, the built-in that performs fixed-priority arbitration among edge, contact and track maneuvers in the device in Fig. 1(a). Arbiter is implemented in EMWF 1.0 with generic built-ins as described in [19]. Table 1

also lists superposition and voter activities for behavior coordination in robotic applications and push data, pull data and mode change activities for all embedded applications. These built-ins are provided by EMWF 2.0.

As pointed out by [19], we can modify a workflow-based device by changing the workflows in it and/or by using different resources for activities. As an example, we can turn the vacuum cleaner in Fig. 1(a) into a navigation component by changing its workflow graph and into a toy sumo by replacing the "back and random move" activity with a "move back and hit" activity. We will return shortly to give another example to illustrate the merit of the workflow approach in this respect.

2.2 Messaging and Real-Time Capabilities

Many UCAADS applications rely on multiple computers, wire and wireless networks, local and remote sensors and control devices, and mobile and fixed user interfaces. Fig. 1(b) shows an example. The block diagram is from [21]. It contains some of the tasks in an experimental mobile robot called Pygmalion. If the robot were workflow based, these tasks would be implemented by workflows. According to [21], the bumpers drivers and speed controller run at 1 KHz, and the position controller runs at 50 Hz. Obstacle avoidance makes use of the vision system and laser range finder. Such a robot may also have a speech recognition system that can capture and interpret voice commands and a speaker location system that can pinpoint the speaker. Navigation and obstacle avoidance tasks may run on a processor while the vision system and speech related tasks run on a separate processor or processors, and speed and position control tasks run on yet another processor.

To support applications exemplified by Pygmalion, we extended EMWF 1.0 with a low-level message passing mechanism that implements push data and pull data built-ins in Table 1 and provides the low-level support essential for end-to-end distributed workflow management. Section 4 will describe the mechanism.

Many tasks shown in Fig. 1(b) have real-time requirements: Obstacle avoidance and position control are examples. They must complete on a timely basis for the robot to move smoothly at a required speed without bumping into obstacles. For such devices, a defect of existing workflow management systems, including EMWF 1.0, as well as existing middleware for robotic applications (e.g., [22]), is their lack of adequate real-time support. This is why we extended EMWF 1.0 with an end-to-end scheduler at the higher level and a message scheduler at the lower-level. Together with the priority-based CPU scheduler provided by the workflow manager in EMWF 1.0, the two-level scheduler enables the extended EMWF to support many well known end-to-end real-time scheduling strategies.

2.3 Workflow Communication

EMWF 1.0 supports event-driven, sequential workflows, but not state machine workflows. More seriously, it lacks flexible, easy-to-use facilities for invocation of workflows by workflows and data exchanges between workflows. These limitations prevent us from using it for complex devices such as iNuC (intelligent nursing cart) [8]. iNuC is a mobile system of medication administration and record keeping tools for nurses. It is self-contained: During network and hospital-wide server outage, the cart can operate stand-alone. The major component tools and system modules include

a graphical user interface (GUI), authentication and authorization module (AaA), work-time manager (WTM), intelligent monitor, alert and notification (iMAN), locker interlock mechanism (LIM), record keeper (RK), data refresher (DR) and cart user event log (CUEL). The beta version iNuC 1.0 has the traditional hardwired structure, is written in C, and runs on Microsoft Windows XP embedded.

A hospital typically needs not only full-service carts like iNuC, but also basic mobile units (BaMU). A BaMU works collaboratively with a per-patient-ward mulit-user medication station (MUMS) server [23] and relies on the server for many functions, including AaA, WTM, and iMAN functions. Building a BaMU by modifying the way modules are integrated in iNuC 1.0 would take considerably more effort than building it from the workflow-base version, called iNuC 1.5.

iNuC 1.5 runs on Microsoft Windows Workflow Foundation (WF) in .NET [11]. The left half of Fig. 2 depicts its structure. Details in the diagram are not important for our discussion here. It suffices to note that the behavior of the cart and its interaction with the user depend almost solely on the iNuC state machine workflow. We can change an iNuC into a BaMU by replacing the iNuC state machine workflow with a BaMU state machine workflow. On WF, such a reconfiguration can be done dynamically without having to restart the cart.

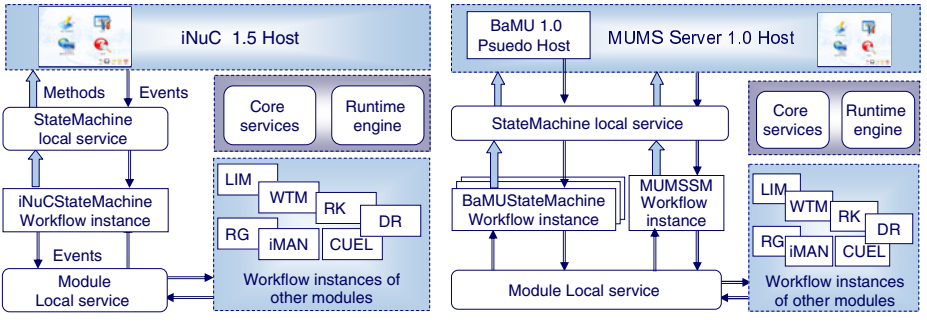


Fig. 2. Workflow-based architecture of intelligent nursing cart and station software

The right half of Fig. 2 shows the workflow-based structure of a multi-user medication station server that works with BaMU to enforce bar-code controlled medication dispensing. The server can support a few remote sessions via BaMU concurrently. It may have multiple BaMU state machine workflow instances interacting via a local service (i.e., a communication facility) with BaMU pseudo host and through it with GUI's running remotely on basic mobile units. Because the workflow runtime engine does the difficult work of managing their execution, we can easily reuse component modules to built different systems.

Before moving on, we note that another advantage of workflow-based design is that the definitions of the workflows and service interfaces of a device specify clearly the device behavior and make the dependencies between components explicit. We can use the definitions as executable behavior specifications to simulate the device and user-device interactions for the purpose of assessing the usability and performance of the device as soon as the specification of the device is available. The simulation environment described in [24] is for this purpose.

WF has many shortcomings for applications such as iNuC, BaMU and Pygmalion, however. Running in .NET environment means not only large system resource demands, but also less than ideal response times for time critical functions. To provide real-time scheduling capabilities requires replacing the default scheduling service by a custom one. This can be done in principle, but a custom scheduler in .NET environment is unlikely to give the developer control on scheduling to the degree necessary for many real-time applications. By providing services for workflow communications, EMWF 2.0 aims to provide the advantages of WF for embedded applications without its shortcomings.

3 Overview of EMWF 1.0

The embedded workflow definition language supported by EMWF 1.0 is called SISARL-XPDL [19]. It is an extended subset of the WfMC standard XPDL 2.0 [13]. The extension includes the built-ins activities listed in the last row of Table 1. It also includes Period and ExtendedAttributes. These elements enable the developer to specify which workflows are real-time, what their execution rates are, and what custom scheduling policy is to be used if the workflows are not to be scheduled by default on rate-monotonic basis. The SISARL-XPDL parser first translates extension elements into standard XPDL 2.0 elements and then compiles XPDL 2.0 definitions into executable workflow scripts. Workflow scripts are stored in .wfs files.

3.1 Engine Manager, Workflow Manager and WLA and ALA Engines

Fig. 3(a) shows the general structures of a workflow-based embedded device running on EMWF 1.0. The application components are shown as workflow instances. Major components of EMWF 1.0 are engine manager, workflow manager and workflow processor. The engine manager manages the configurations of the engine and application workflows. The workflow manager processes the workflow scripts, and the workflow processor executes activities according to the scripts.

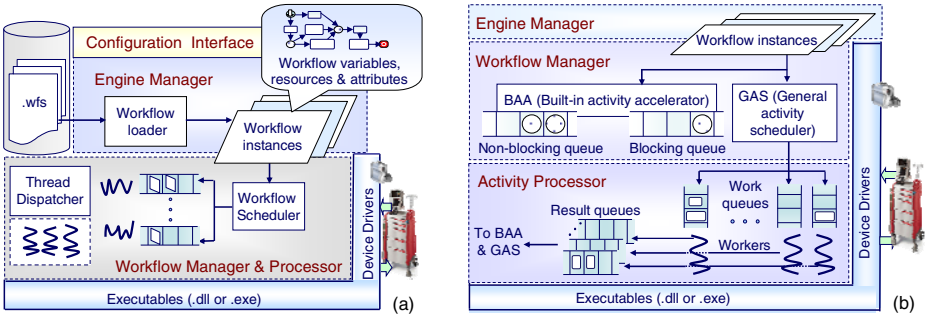


Fig. 3. Parts (a) Workflow-based device with WLA engine and (b) structure of ALA engine

The engine manager is responsible for handling user requests and managing their accesses to workflow-related definitions and optional contextual information. The

developer can tune the engine via the configuration interface. Configuration parameters include the maximum numbers of threads and priority levels, and the finest resolution of timers. During initialization, the engine manager initializes and configures the engine. It then loads the .wfs files needed by all the applications for all operation modes and adaptation into memory. This enables the workflow manager to allocate memory dynamically for all instances of activities and workflows during initialization in order to keep the memory footprint of the applications small. A negative consequence is that when .wfs files are added and removed, the engine must be restarted for the configuration changes to take effect.

EMWF 1.0 offers two different multi-threaded workflow processors on Microsoft Windows CE and XP Embedded. They are the WLA (workflow-level assignment) engine and the ALA (activity-level assignment) engine. The lower half of Fig. 3(a) depicts the structure of the WLA engine. In this case, each thread is dedicated to execute a workflow: The workflow manager attaches a thread to each workflow instance when it initializes the instance and schedules the thread to execute all activities in it. The thread inherits the priority of the workflow instance.

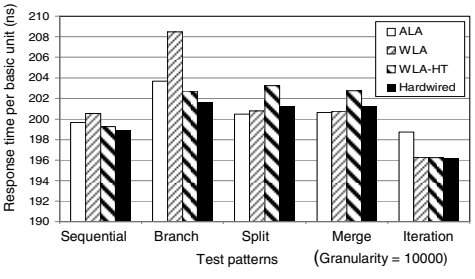
Fig. 3(b) depicts the structure of an ALA engine. The term general activity in the figure refers to activities provided by the developer, i.e., not built-ins of the engine. An ALA engine uses worker threads to execute activities as work items: The workflow manager maintains a FIFO queue per priority for queuing work items and assigns at least a thread per queue. The thread (or threads) serving a queue executes at the priority of the queue. Threads in the workflow manager and processor interact in more or less the leader/followers pattern. Worker threads are followers. For a device that has no blocking built-in activities, the workflow manager may have just one leader thread. The leader processes workflow scripts, wraps ready (i.e., enabled) general activities as work items and inserts them in priority queues according to their priorities to be executed by follower threads, and supervises their completion. With a few exceptions, built-ins are simple. The leader executes itself built-ins as they become enabled, which in turn leads to more general activities be ready and queued. These functions of the leader are depicted as general activity scheduler and built-in activity accelerator in Fig. 3(b).

3.2 Relative Merits

Since a thread assigned to a workflow in a WLA engine executes both general activities and built-ins in it, most of the transitions between activities incur no context switch. In an ALA engine, however, every transition from one general activity to another incurs at least one context switch.

The current versions of WLA engines do not support varying priority within workflows, just like Microsoft .Net WF, which also uses the WLA strategy. Priority increment of an activity with respect to the base priority of the workflow containing it is ignored by the engine. In contrast ALA engine executes activities at their own priorities and thus supports varying workflow priority naturally. Coherent time order for all tasks within a device is often expensive to achieve. Using an ALA engine, this can be accomplished with no additional cost by having the engine use a single thread to handle all timing events.

We have measured the runtime performance of WLA and ALA engines using several benchmark workflow-based workloads running on a 3.4 GHz Pentium 4 and Windows CE 6.0 platform. Each workflow-based workload is characterized by the number of workflows, a test pattern and the granularity of activities. For each workflow-based load, we also ran equivalent hardwired code on Windows CE 6.0 without the engine. The difference between the response times for the two versions gives us an estimate of the runtime overhead introduced by the engines. The chart in the left half of Fig. 4 shows the kind of performance data obtained from this study. In this case, all activities are software activities with the same granularity. (The function for each activity calls a random number generator 10,000 times.) The WLA-HT engine is an enhanced WLA engine which uses a helper thread to create workflow instances. We can see that when activities are of sufficiently large granularity, as in the case shown here, runtime overheads of ALA and WLA-HT engines are acceptable. The disadvantage of ALA engine in terms of context switches becomes evident when activities are so small that the number of extra context switches is in order of 1000 per workflow. Similar measurements performed on Windows XP Embedded points to similar conclusions. Details on this study can be found in [19].



Performance under Hartstone PH-series load

	Hardwired	WLA	ALA
Task 1	26 (2, 33)	31 (2, 29)	27 (9, 18)
Task 2	51 (8, 43)	62 (4, 58)	59 (17, 42)
Task 3	93 (44, 49)	125 (8, 117)	116 (28, 88)

Task routine: Array processing

Task 1	35 (2, 33)	31 (1, 30)	31 (3, 28)
Task 2	69 (4, 65)	62 (1, 61)	62 (7, 55)

Task routine: Standard function calls

Fig. 4. Performance data from [19] and from a test using Hartstone workload

The tables in the right half of Fig. 4 list data on engine performance taken by running Hartstone PH-series [25] workload on a 1.73 GHz Genuine Intel CPU and Windows Standard XP SP3 with 1-ms timer resolution. The workload contains 5 periodic, harmonic tasks, and they were scheduled rate-monotonically. Again, we ran an equivalent hardwired code without the engine: In the hardwired version, each task is a single thread that executes the task routine periodically. In the workflow version, the routine is executed as an activity by the engine. We used two task routines: One processes a small integer array and the other makes standard function calls. The former is shorter (i.e., gives us a smaller granularity) than the latter.

We started a series of tests by setting the highest rate (i.e., the rate of Task 5) at 32 Hz and for subsequent tests, increased the rate until it reaches the highest possible rate of 500 Hz. For each test, we ran the tasks for 10 seconds and recorded for each task the number M of misses (i.e., the number of periods in which the task ran but completed late) and number S of skips (i.e., the number periods in which the task was not scheduled). The number T of timing faults is the sum $M + S$. We use the notation T (M, S) to capture these numbers.

When the highest rate was 32Hz and task routine was standard function calls, Task 5 with rate 32Hz had 4(2, 2) and 75(9, 66) timing faults in 320 periods when the tasks were executed by the WLA engine and ALA engine, respectively. On the ALA engine, each task incurs a context switch each period, while there is no context switch on the WLA engine and for the hardwired code. This is a reason that Task 5 incurred a larger number of timing faults when executed on the ALA engine. All other tasks incurred no timing faults. When the task routine was array processing, all tasks, including Task 5, had no timing fault. When the highest rate was 500Hz, the system was overloaded. The tables in Fig. 4 list the numbers of timing faults of lower priority tasks, showing that the engines performed comparably under such condition.

4 Extensions in EMWF 2.0

We presented earlier rationales for extending EMWF 1.0 with two kinds of extensions. First, messaging passing and end-to-end scheduling mechanisms are essential parts of workflow management for time-critical distributed and networked applications. These mechanisms have already been added to EMWF 1.0 on Windows XP Embedded. The second kinds of extensions include capabilities and tools that are needed to make EMWF 2.0 not only a middleware ideally suited for a wide range of user-centric automation and assistive devices from simple devices like automatic vacuum cleansers to complex ones like iNuC and Pygmalion, but also an excellent design, development and evaluation environment for them.

4.1 Messaging and End-to-End Scheduling Mechanisms

Fig. 5 (a) illustrates how push data and pull data built-in activities are implemented. Without loss of generality, suppose that there are two workflows: Workflow 1 contains a push data built-in while Workflow 2 contains two pull data built-ins. When encountering a push data activity, the workflow manager dispatches a Send operation to move the data to be sent into a send buffer and queues a Send work item in one of the prioritized work queues according to the priority of the Send operation. The work item is executed by a worker thread at the priority of the queue. Depending on whether the receiving workflow runs locally or remotely, the Send work item either invokes an IPC or a Winsock send API function to move the data from the send buffer to the receive buffer. This and other data transfers are depicted by dashed arrows. Once arrived, the data waits in the receive buffer until the workflow manager of the receiving workflow encounters a pull data activity, depicted as a box with a block arrow pointing into the box. The manager then queues a Receive work item to move the data from the receive buffer to the space of the receiving workflow. Specific work to be done by Receive is application dependent. The developer can specify how it is to be done via a parameter of Receive.

Rather than a library of schedulers for different types of applications, EMWF provides a general two-level, end-to-end scheduling mechanism with which many well-known and commonly used real-time scheduling schemes can be easily implemented. Fig. 5(b) shows the structure of the mechanism for ALA engine. The description for it is by and large applicable to the case of the WLA engine.

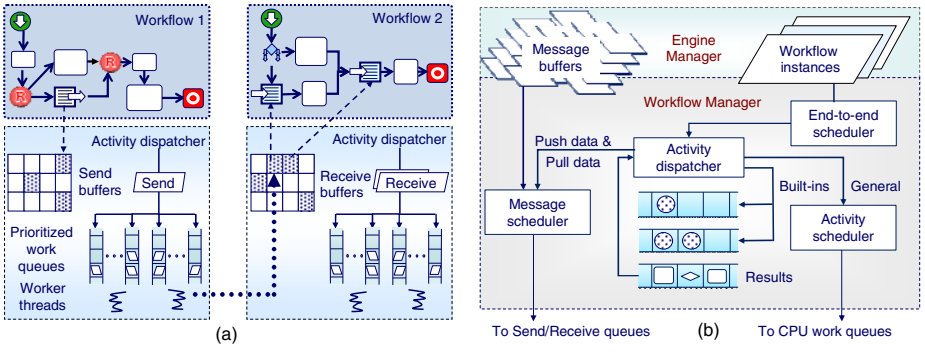


Fig. 5. Real-time extensions (a) Send/Receive mechanism and (b) two-level scheduler

The assumption here is that the system is statically configured: In other words, components of each application are partitioned and assigned to processors (and other types of resources) at initialization time. They are migrated among processors only when reconfiguration becomes necessary. The extended attributes of each workflow instance contains information on the CPU used for each software activity and resources required by external activities. If the workflow has real-time requirements, the extended attributes also specify a finite end-to-end deadline, estimated worst-case execution time of each activity and so on. The end-to-end slack time of a workflow instance is the difference between its end-to-end deadline and the total worst-case execution time of the longest chain of activities in the workflow.

The workflow manager calls the high-level end-to-end scheduler when a new workflow instance becomes ready for execution. Currently, the only task performed by the scheduler is to distribute the available end-to-end slack time of each workflow (or chain of workflows) to activities or chains of activities according to specified algorithm(s). Both low-level schedulers (i.e., activity scheduler for CPU scheduling and message scheduler for network traffic) support fixed priority scheduling.

EMWF 1.0 requires the developer to partition the application(s), assign processors (and resources) to individual partitions and provide algorithms for distributing end-to-end slack and priority assignments for CPU and network scheduling. EMWF 2.0 will provide admission control and resource management tools to support this work.

4.2 Service Interfaces

Earlier in Fig. 2, we introduced the term local service without explaining what it means and what the components bearing this name do and why EWMF 2.0 needs to provide similar support. Simply put, from the point of view of a workflow-based application, a local service (e.g., StateMachineService or ModuleService in Fig. 2) of Window .NET WF [11] is a set of interface functions that are defined and implemented by the developer of the application. To illustrate, the right half of Fig. 6 shows the interface functions of the local services in Fig. 2. Workflow instances in the application communicate with non-workflow component(s) and with each other using these interface functions.

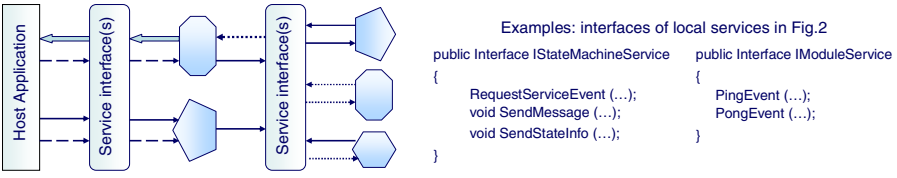


Fig. 6. Interaction via service interfaces

EMWF service interfaces resemble closely to local services of WF. The kind of assistance EMWF 2.0 will provide to service interfaces is similar to what WF does for local services: In particular, each service interface is identified by its type. After the developer has declared a service interface type, implemented a service interface of the type and have the application created and registered the service interface with the workflow management runtime during initialization time, workflow instances in the application can query the workflow manager for functions provided by the service and make use of the functions for communication and invocation.

The diagram in Fig. 6 illustrates ways workflow instances and non-workflow components communicate and interact, making use of service interfaces. Rectangular shapes represent service interfaces, host applications and non-workflow components. Polygons represent workflow instances of the same or different type. Bold arrows represent callbacks and simple arrows represent raises and deliveries of events.

Specifically, workflow instances invoke each other and deliver results to each other by raising events. In essence, the system of service interfaces serves as a router, routing each event raised by a workflow instance to one or more workflow instances as specified by the parameters of the raise-event interface function. In a distributed system, the workflow manager helps to track the locations of all workflow instances and thus relieves the developer from the burden of this work. In addition to events, workflow instances can communicate with the host application and other non-workflow components via callbacks.

We note that when a caller raises an event to invoke a workflow instance, the event handler is executed by the thread dispatched to execute the instance. When the workflow instance responds to a caller via a callback function, the function is also executed by this thread. It is important to follow the principle of this pattern when the caller must be responsive. This is the case of the host in iNuC. The host application contains a single thread. After initialization, it becomes the GUI thread. Using the communication pattern depicted by Fig. 6, the GUI thread is never tied up doing time consuming work. It is almost always free and ready to respond to user action.

5 Summary

This paper describes the current status and future plans for EMWF. The workflow management system enables diverse embedded devices to be built from activities and workflows components. Its light-weight engine integrates the components at runtime by executing the components in manners specified by the developer. Configurability is one of primary factors that motivated us to build the embedded workflow framework. The proof-of-concept version EMWF 1.0 provides light-weight engines on

Microsoft Windows CE and XP Embedded platforms. They are released under GPL license. This version has many limitations. We are working to remove them while adding the features and capabilities described in the previous section and maturing the middleware into EMWF 2.0.

Acknowledgments. This work was partially supported by the Taiwan Academia Sinica thematic project SISARL.

References

1. Tsai, P.H., Yu, C.Y., Wang, W.Y., Zao, J.K., Yeh, H.C., Shih, C.S., Liu, J.W.S.: iMAT: Intelligent Medication Administration Tools. In: Proc. of IEEE Healthcom (July 2010)
2. Wang, W.Y., Zao, J.K., Tsai, P.H., Liu, J.W.S.: Wedjat: A Mobile Phone Based Medication Reminder and Monitor. In: Proceedings of the 9th IEEE International Conference on Bioinformatics and Bioengineering (June 2009)
3. Tsai, P.H., Yu, C.Y., Shih, C.S., Liu, J.W.S.: Smart Medication Dispenser: Architecture, Design and Implementation. Technical Report No. TR-IIS-008-010, Institute of Information Science, Academia Sinica (2008)
4. Chou, T.S., Liu, J.W.S.: Design and Implementation of RFID-Based Object Locator. In: Proceedings of IEEE 2007 International Conference on RFID Technology (March 2007)
5. Hsu, C.F., Liao, H.Y.M., Hsiu, P.C., Shih, C.S., Kuo, T.W., Liu, J.W.S.: Smart Pantries for Homes. In: Proceedings of IEEE International Conference on SMC (September 2006)
6. Forizzi, J., DiSalvo, C.: Service Robots in Domestic Environment: a Study of Roomba Vacuum in the Home. In: Proc. of ACM/IEEE International Conference on HRI (March 2006)
7. Kaneshige, Y., Nihei, M., Fujie, M.G.: Development of New Mobility Assistive Robot for Elderly People with Body Functional Control. In: Proceedings of IEEE/RAS-EMBS (February 2006)
8. Tsai, P.H., Chuang, Y.T., Chou, T.S., Shih, C.S., Liu, J.W.S.: iNuC: An Intelligent Mobile Medication Cart. In: Proceedings of the 2nd International Conference on Biomedical Engineering and Informatics (October 2009)
9. SpeciMinder hospital delivery robot, http://www.youtube.com/watch?v=lJ7RnTAYZ-8
10. TUG, pharmacy delivery robot, http://hfrp.umm.edu/tug/tug_main.htm
11. Bukovics, B.: Pro. WF: Windows Workflow Foundation in .Net 4.0. Apress (2009)
12. WfMC: Workflow Management Coalition, http://www.wfmc.org/
13. XPDL (XML Process Definition Language) Document (October 2005), http://www.wfmc.org/standards/docs/TC-1025_xpdl.2.2005-10-03.pdf
14. BPEL (Business Process Execution Language), http://en.wikipedia.org/wiki/BPEL
15. Open Source Java XPDL editor, http://www.enhydra.org/workflow/jawe/index.html
16. Enhydra Shark, http://forge.objectweb.org/projects/shark
17. Pajunen, L., Chande, S.: Developing workflow engine for mobile devices. In: Proc. of IEEE International Enterprise Distributed Object Computing Conference (2007)

18. Hackmann, G., Haitjema, M., Gill, C., Roman, G.C.: Silver: A BPEL workflow process execution engine for mobile devices. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 503–508. Springer, Heidelberg (2006)
19. Chou, T.S., Chang, S.Y., Lu, Y.F., Wang, Y.C., Ouyang, M.K., Shih, C.S., Kuo, T.W., Hu, J.S., Liu, J.W.S.: EMWF for Flexible Automation and Assistive Devices. In: Proceedings of IEEE RTAS (April 2009)
20. Chang, S.Y., Lu, Y.-F., Kuo, T.W., Liu, J.W.S.: The Design of a Light-Weight Workflow Engine for Embedded Systems. In: Proceedings of RTSS Workshop on Software and Systems for Medical Devices and Services (December 2007)
21. Brega, R., Tomatis, N., Arras, K.O.: The Need for Autonomy and Real-Time in Mobile Robotics: A Case Study for X0/2 and Pygmalion. In: Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (October 2000)
22. Robot Standards and Reference Architecture, `http://wiki.robot-standards.org/index.php/Middleware`
23. Liu, J.W.S., Shih, C.S., Tan, C.T., Wu, V.J.S.: MeMDAS: Medication Management, Dispensing and Administration System. In: m-Health Workshop, IEEE HealthCom 2010 (2010)
24. Chen, T.Y., Chen, C.H., Shih, C.S., Liu, J.W.S.: A Simulation Environment for the Development of Smart Devices for the Elderly. In: Proceedings of IEEE International Conference on Systems, Man and Cybernetics (October 2008)
25. Weiderman1, N.H., Kamenoff, N.I.: Hartstone Uniprocessor Benchmark. Journal of Real-Time Systems (December 1992)

An Investigation on Flexible Communications in Publish/Subscribe Services

Christian Esposito[2], Domenico Cotroneo[1], and Stefano Russo[1,2]

[1] Dipartimento di Informatica e Sistemistica (DIS)
Università di Napoli Federico II, Napoli, 80125 - Italy
[2] Laboratorio CINI-Item "Carlo Savy"
Consorzio Interuniversitario Nazionale per l'Informatica (CINI),
Napoli 80125 - Italy
{christian.esposito,cotroneo,sterusso}@unina.it

Abstract. Novel embedded and ubiquitous infrastructures are being realized as collaborative federations of heterogeneous systems over wide-area networks by means of publish/subscribe services. Current publish/subscribe middleware do not jointly support two key requirements of these infrastructures: timeliness, *i.e.*, delivering data to the right destination at the right time, and flexibility, *i.e.*, enabling heterogeneous interacting applications to properly retrieve and comprehend exchanged data. In fact, some middleware solutions pay more attention to timeliness by using serialization formats that minimize delivery time, but also reduce flexibility by constraining applications to adhere to predefined data structures. Other solutions adopt XML to improve flexibility, whose redundant syntax strongly affects the delivery latency.

We have investigated the consequences of the adoption of several lightweight formats, which are alternative to XML, in terms of flexibility and timeliness. Our experiments show that the performance overhead imposed by the use of flexible formats is not negligible, and even the introduction of data compression is not able to manage such issue.

Keywords: Flexibility, Timeliness, Serialization Formats, Pub/Sub Services.

1 Introduction

Typically, embedded systems have been architected according to a "*closed world*" perspective: a series of computing machines were interconnected by dedicated networks, but with limited, or no, cooperation with the outside world. Therefore, frameworks that have to carry out complex control activities, such as the Air Traffic Management (ATM) or the Power System Control (PSC), have been fragmented into "*islands of automation*", *i.e.*, they are composed by several autonomous and independent systems, each one in charge of controlling an isolated portion of the overall framework, but with no reciprocal cooperation. Inefficiencies of such traditional perspective and recent developments in networking are causing an evolution of embedded systems, leading to the so-called Large scale

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 204–215, 2010.

Complex Critical Infrastructures (LCCI). Such infrastructures adopt a federated, "*open world*" architecture, *i.e.*, LCCI consist of dynamic Internet-scale hierarchies/constellations of interacting heterogeneous systems, which cooperate to carry out critical functionalities. Many of the ideas behind LCCI are increasingly "in the air" in several current projects that aim to devise innovative critical embedded systems. For example, EuroCONTROL has funded a project, called "*Single European Sky ATM Research*" (SESAR)[1], to develop the novel European ATM framework as a seamless infrastructure allowing control systems to cooperate in order to better handle the growing avionic traffic.

This novel perspective is also enforcing the integration of several different kinds of IT infrastructures, which are usually strictly distinct and separated from critical embedded systems. A practical example is provided by another EU project, called "*Total Airport*"[2], whose scope is the integration of all subsystems for land-side and air-side activities and their information flows. Specifically, critical embedded systems and the relative federation middleware under development within the context of SESAR are going to be integrated with all the IT components for airport management, and also with ubiquitous systems for safe and secure passenger and luggage management, so to realize a seamless "*door-to-door*" control (*i.e.*, from when entering the departing airport until leaving the arriving airport). *E.g.*, passengers can access to certain ATM information via their smart phones to know the status or schedule of their flights, to track their luggages, to locate themselves within the airport map or even to receive commercial ads. Such innovative embedded and ubiquitous systems require several non-func-tional requirements to be satisfied, among which there is flexibility, *i.e.*, interacting entities must be able to comprehend each other even if they do not know the structure applied by the data source to the exchanged messages. The widely-adopted middleware solutions in federating heterogeneous systems are the ones that adopt the publish/subscribe interaction model, called pub/sub services, due to its intrinsic decoupling properties that enforce efficient and scalable data dissemination. However, most of these solutions adopt serialization formats that negatively affect the flexibility offered by the middleware. A widely-adopted solution to resolve such drawback is to adopt XML as serialization format; however, this is not a winning choice due to the high performance overhead of XML-based communications.

In this paper we study the use of two lightweight flexible formats in Subsection 3.2, and discuss their performance in Section 4. Our experiments have shown that flexibility is always obtained at high expenses of performance. Therefore, we investigate in Section 6 the effects of introducing data compression (briefly described in Section 5) to reduce the performance drawbacks of flexible formats. The conducted measurement campaigns have revealed that data compression is not able to provide a considerable improvement of performance, but is able to better tolerate message losses.

[1] www.eurocontrol.int/sesar/

[2] www.eurocontrol.int/eec/public/standard_page/EEC_News_2006_3_TAM.html

2 Problem Statement

Large-scale systems are rarely built ex-novo, but it is more probable that they are developed starting from already-existent legacy systems by using a middleware and other proper abstractions to federate them. Federating legacy systems, built by different companies at different times and under different regulation laws, raises the so-called *Data Exchange Problem* [1]. Specifically, let us consider an application, namely A_{source}, which is a data source and is characterized by a given schema, indicated as S_{source}, for the produced data, and another one, namely A_{dest}, which is a data destination and is characterized by a given schema, indicated as S_{dest}. When the two schemas diverge, a communication can take place only if a mapping M between the two schemas exists. This allows the destination to comprehend the received messages and to opportunely use them within its application logic. When the two schemas are equal, the mapping is simply the identity. On the contrary, when several heterogeneous legacy systems are federated, it is reasonable to have diverging data schemas, and the middleware solution used for the federation needs to find the mapping M and to adopt proper mechanisms to use it in the data dissemination process. Moreover, the communication pattern adopted in collaborative infrastructures is not one-to-one, but one-to-multi or multi-to-multi. So, during a single data dissemination operation, there is not a single mapping M to be considered, but several of them, *i.e.*, one per each destination.

EUROCONTROL has tried to find a solution to this problem by specifying a standard schema for the flight data exchanged among ACCes, called ATM Validation ENvironment for Use towards EATMS (AVENUE)[3]. However, even if neglecting the strong additional overhead introduced by mapping to/from the standard data structure both at publisher and subscriber side, this workaround does not completely resolve the issue. In fact, over the time, a standard data format is likely to be changed in order to address novel issues or to include more data (in fact, in the last three years AVENUE has been updated several times). However, not all the systems may be modified to handle new versions, so there may be systems with different versions. This brings back the Data Exchange Problem when a publisher produces events with a certain version of the standard data structure and subscribers can comprehend only another versions.

To completely resolve such issue, the viable solution is to realize a *flexible communication*: the data source does not care about the schemas of the receivers. On the other hand, data destinations are able to introspect the structure of received messages and use such information to properly feed data instances.

3 Serialization Formats in Publish/Subscribe Middleware

Pub/sub services are very appealing to efficiently interconnect several systems due to their intrinsic decoupling properties that promote scalability [2]. In fact, a recent OMG specification for pub/sub services, defined by OMG and called Data

[3] www.eurocontrol.int/eec/public/standard_page/ERS_avenue.html

Distribution Service (DDS) [3], has been chosen by EUROCONTROL as the reference technology for its novel European ATM framework under development within the context of the SESAR project. To study if pub/sub services are able to provide flexible communication to address the Data Exchange Problem presented in the previous section, it is crucial to analyze the features of serialization formats that they adopt. Therefore, in the following Subsection 3.1, we present the main serialization formats adopted in the current pub/sub services, and discuss their pros and cons to support flexible communication. Instead, in Subsection 3.2 we introduce new serialization formats as suitable alternatives.

3.1 Current Serialization Formats

CDR. Some of the available pub/sub services adopt serialization formats that can be defined as *binary*, and a practical example is the *Common Data Representation* (CDR) [4], adopted by all products compliant to DDS specification. Binary formats are based on a positional approach: serialization, and relative deserialization, operations are performed according to the position occupied by data within the byte stream. To better explain how binary formats work, let us consider a publisher and subscriber exchanging a certain data instance. The publisher goes through all the fields of the give data instance, converts the content of each field in bytes, and stores it in a byte stream, treated as a FIFO queue. On the subscriber side, the application feeds data instances with information conveyed by received byte streams. Specifically, knowing that the serialization of the first field of type T requires a certain number, namely n, of bytes, the subscriber extracts the first n bytes from the byte stream. Then, it casts such n bytes in the proper type T and assigns the obtained value to the field in its own data instance. Such operation is repeated until the entire data instance is filled.

CDR does not support a flexible communication. In fact, the ability of the subscriber to comprehend received messages, *i.e.*, to obtain original data instances starting from received byte streams, is coupled to the knowledge of the data structure at the publisher side. On the other hand, since only instance content is delivered throws the network, formats such as CDR exhibit a serialization stream characterized by a minimal size.

XML. When a middleware solution wants to provide flexible communication, it typically uses serializations formats defined as *tree-based*, *i.e.*, they embodies in the serialization stream not only instance content, but also meta-information, organized as a tree, about the internal structure of the data instance. Such meta-information allow decoupling interacting applications from the reciprocal knowledge of the internal structure of the data that they are exchanging. The widely-adopted tree-based format is *XML*, which specifies the structure of data content by a combination of opening and closing tags. In fact, there has been an increasing demand for XML-based pub/sub services, which support flexible document structures and subscription rules expressed by powerful languages, such as XPath and XQuery [5].When using XML, the publisher transforms a data

instance in an XML document by placing the content of its fields between tags with the same name of the given field. Such document is further converted into a byte stream and delivered to the subscriber, which uses such XML document to feed a data instance by assigning to each field the value between tags equal to the field name.

Adopting XML allows the subscriber to be not aware of the data structure applied at the publisher side since stream structure is no more implicit, but explicit into the tags. So, flexible communication is supported; however, such flexibility is achieved at the expenses of delivery latency. In fact, XML syntax is redundant or larger with respect to binary formats of similar data, and this redundancy may affect application efficiency through higher transmission and serialization costs.

3.2 Lightweight Flexible Serialization Formats

Some application scenarios for collaborative IT infrastructures show a time-critical behaviour: delivering information out of time boundaries could lead to instability (*timeliness*). So, it is important to minimize the transmission latency. For this reason, XML is not the viable solution to support flexibility, but other tree-based formats that exhibit lower performance costs are needed. In literature there are available other tree-based formats that are simpler than XML while maintaining its flexibility guarantees. In fact, they have been specifically designed as a data interchange format, not a markup language. So, the dimension of the serialized stream is smaller than the one obtainable with XML, because there is no redundant syntax, *e.g.*, no closing tags. In the rest of this subsection, we will present two of these "lightweight" flexible serialization formats.

JSON. Java Script Object Notation (JSON)[4] is a lightweight data-interchange format, based on a subset of the JavaScript Programming Language[5]. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages. It is built on two structures: a collection of name/value pairs, and an ordered list of values. When using JSON, serialization and deserialization is performed as seen in the case of XML, but using a collection of name/value pairs allows saving bytes in the serialization stream.

YAML. YAML Aint Markup Language (YAML)[6] is a data serialization format that takes concepts from languages such as XML, C, Python, Perl, as well as the format for electronic mail as specified by RFC 0822[7]. Its syntax is relatively

[4] www.json.org/index.html
[5] Standard ECMA-262, 3rd Edition - December 1999. More details available at www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf
[6] www.yaml.org
[7] www.ietf.org/rfc/rfc0822.txt

straightforward, with data structure hierarchy maintained by outline indentation, which facilitates easy inspection of the data structure. YAML can waste bytes, since each space of the indentation must be translated as a character in the serialization stream. To overcome this problem, its possible to use a compact version of YAML by replacing indentation with brackets.

Comparison. Both JSON and YAML share very similar syntax; however, they also exhibit certain differences. JSON is trivial to generate and parse. It also uses a lowest common denominator information model, ensuring any JSON data to be easily processed. On the other hand, YAML is more complex to generate and parse. In addition, YAML ventures beyond the lowest common denominator data types, requiring more complex processing.

4 Experimental Evaluation of Serialization Formats

The goal of this section is to compare all the serialization formats illustrated in the previous section by analyzing their quality in terms of two measures: *serialization efficiency*, *i.e.*, how many bytes in the serialization stream are added to the instance content, and *latency*, *i.e.*, how much time is needed to exchange data instances from a publisher to a subscriber.

4.1 Experiment Setup

We have realized a prototype to exchange data instances that are structured according to the AVENUE type, which is characterized by a complex structure made of about 30 nested fields and a size of almost 100 KB. In addition, we have used an implementation of DDS, provided by RTI, as pub/sub service to exchange messages between a publisher and a subscriber. Moreover, we have implemented a component, named Parser and placed it between the application and the middleware, which takes data instances from the application and returns byte streams to the middleware and vice versa. Within such component, we have embodied the following parsers: (*i*) an in-house developed parser for CDR and YAML, (*ii*) XERCES parser[8], using both DOM and SAX, for XML, and (*iii*) JOST parser[9] for JSON.

The experiments conducted to evaluate latency adopt a "ping-pong" pattern, illustrated in Figure 1. Specifically, the publisher feeds a data instance with randomly-generated content and passes it to the parser, which returns a byte stream to the middleware for disseminating it. On the subscriber side, the middleware receives the byte stream, which is passed to the subscriber application. Then, the subscriber application immediately makes a copy of the received stream and sends it back to the publisher, which receives the original message after the stream passed through the parser component. Along the path from the

[8] www.apache.org/xerces

[9] ddsbench.svn.sourceforge.net/viewvc/ddsbench/trunk/jost/

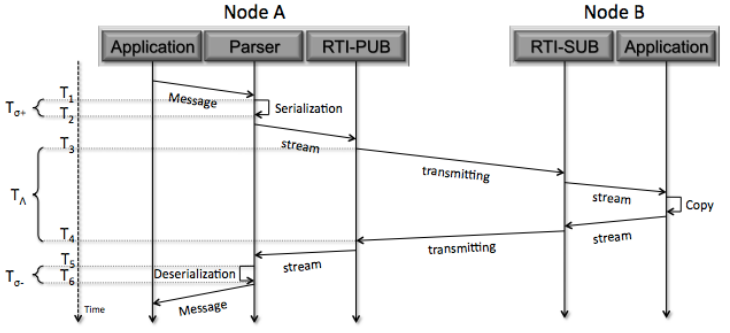


Fig. 1. Serialization and Deserialization operated according to YAML

publisher to the subscriber and backward, we take several timestamps in order to characterize the achievable latency in terms of the following three contributions:

1. *serialization time* $(T_2 - T_1)$, *i.e.*, time spent by the parser to convert data instances into byte streams;
2. *deserialization time* $(T_6 - T_5)$, *i.e.*, time taken by the parser to convert byte streams into data instances;
3. *delivery time* $(T_4 - T_3)$, *i.e.*, time needed for a message to go from the publisher to the subscriber and backward.

4.2 Results

Figure 2 illustrates the outcomes of our experimental campaign. It is not unexpected that CDR presents the highest efficiency, but it is surprising how bad the tree-based formats perform, exhibiting a mean efficiency of 0,0854, which is pretty far from the efficiency achieved by CDR. Among the tree-based formats, the ones with the better efficiency are JSON and the compact version of YAML (respectively with an efficiency equal to 0,114 and 0,105), while the worst efficiency has been registered for XML (*i.e.*, the efficiency is equal to 0,0678). Efficiency affects the measured delivery time, shown in Figure 2D, which is higher when using data format with lower efficiency due to the higher number of bytes to exchange. Figures 2B and 2C, which respectively illustrate serialization and deserialization time, are more interesting. They show that CDR has the best performances due to the simplicity of the parsing operations. With respect to the tree-based formats, JSON and compact version YAML realize again the best performance considering serialization time, while full version of YAML and XML[10] have the worst one. Considering the deserialization, Full and Compact versions of YAML present performance closer to XML with SAX, while XML with DOM achieves the highest measured deserialization time, and JSON presented the lowest deserialization time.

[10] Even if we used SAX in our experiments, DOM is still used to carry out the serialization duties.

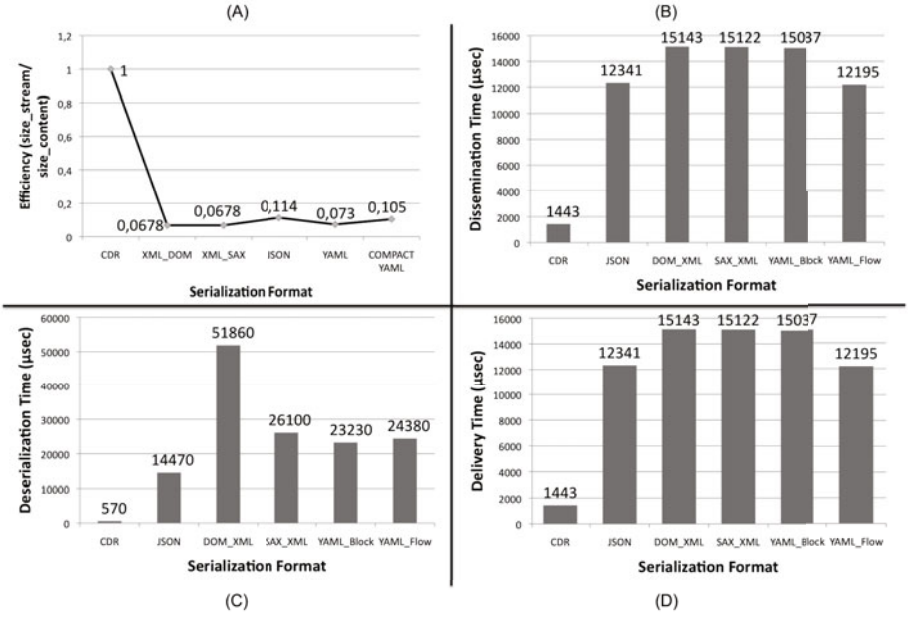


Fig. 2. Experimental Results analyzing the following metrics: (*A*) serialization efficiency, (*B*) serialization time, (*C*) deserialization time and (*D*) delivery time

5 Data Compression

Previous experimental results have clearly proved the considerable performance overhead implied by tree-based formats, making them inapplicable in application scenarios where timeliness is also a key requirement to be satisfied. A possible solution to limit this drawback is to use *data compression techniques* [6] after the parser and before the middleware layer. During the last years, several data compression techniques have been presented by academia or industry. Such techniques can be classified in two distinct classes: *lossy*, *i.e.*, some pieces of information may be lost after decompression, and *lossless*, *i.e.*, pieces of information are never lost after decompression. Since we do not want to incur in any occurrence of data losses, we have preferred techniques belonging to the second class. The most used lossless compression techniques are the following ones: (*i*) optimal coding of Huffman, (*ii*) Lempel-Ziv (LZ) algorithm, and (*iii*) Run-length encoding (RLE). Such techniques are known to achieve between 50% and 30% as compression efficiency (*i.e.*, the compressed stream presents 50% - 30% less bytes than the original stream). If higher compression efficiency is needed, the literature is rich of *hybrid schemas* that combine the previous techniques: (*i*) zlib[11], which adopts the "*DEPLATE*" method to combine LZ and Huffmann

[11] www.zlib.net/

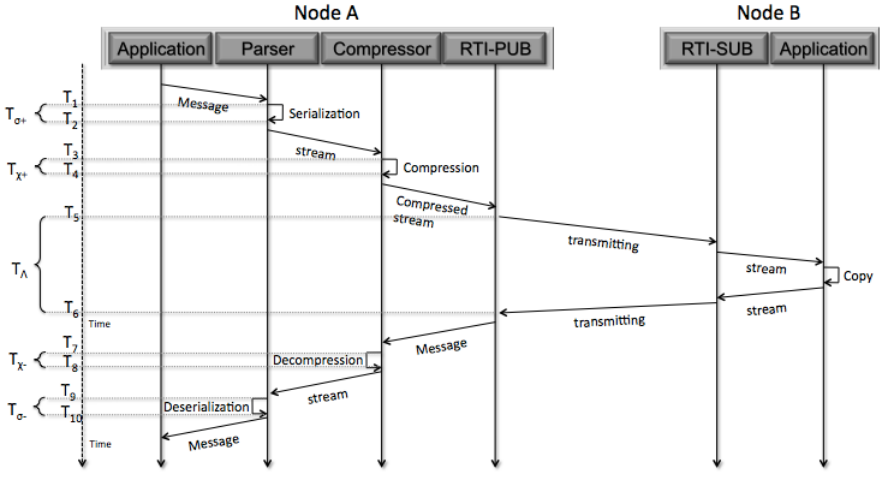


Fig. 3. Serialization and Deserialization operated according to YAML

Coding, (*ii*) bzip2[12], which uses the Burrows-Wheeler block sorting technique and Huffman coding, and (*iii*) Lempel-Ziv-Oberhumer (LZO) algorithm[13].

6 Experimental Evaluation of Serialization Formats with Data Compression

This section presents the two kind of experiments that we have conducted. The first campaign is similar to the one presented in Section 4 and aims at showing the improvement in performance and efficiency when tree-based formats are teamed up with compression techniques. On the other hand, the goal of the second one is to study the behaviour of the developed prototype in a real case scenario that uses Internet as Interconnection network.

6.1 Experiment Setup

We have modified the prototype used in the previous experiments by introducing an additional component, named Compressor, which embodies data compression techniques, as clearly shown in Figure 3. Therefore, we have characterized the achievable performance as in Subsection 4.1 but adding two more contributions:

- *compression time* $(T_4 - T_3)$, *i.e.*, time spent to perform compression;
- *decompression time* $(T_8 - T_7)$, *i.e.*, time elapsed to make decompression.

[12] www.bzip.org/
[13] www.oberhumer.com/opensource/lzo/

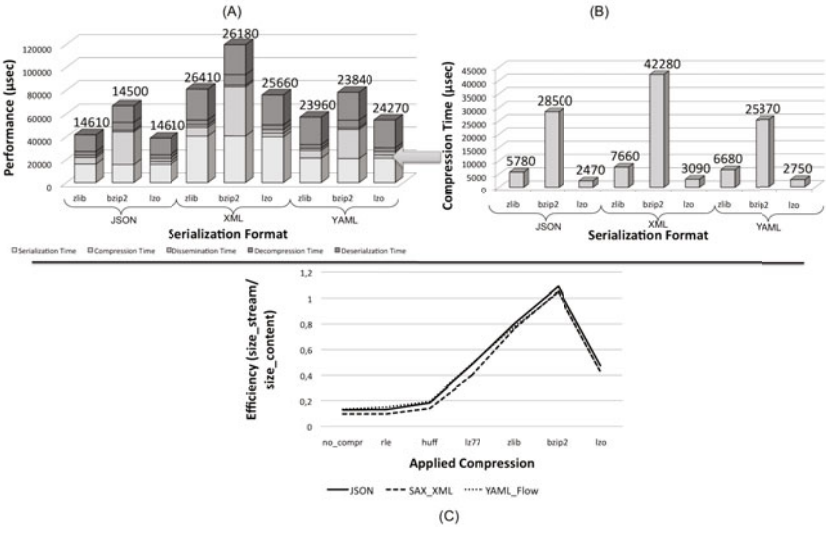


Fig. 4. (A) performance, (B) compression time, and (C) efficiency of first campaign

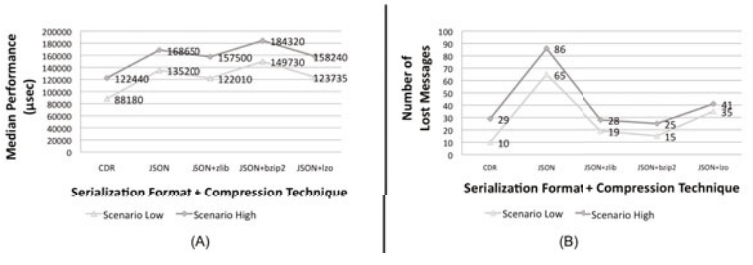


Fig. 5. (A) performance and (B) resiliency of the second experimental campaign

With respect to the campaign using a real case scenario, *i.e.*, applications interconnected by wide area networks as Internet, experiments could not be performed using a realistic testbed, such as PlanetLab[14], because it has not been designed to perform controlled experiments and to achieve reproducible results [7]. Thus, we have chosen to adopt an emulation approach: the publisher and subscriber are executed on distinct machines, interconnected by a network emulator called Shunra Virtual Enterprise (VE)[15], which allows users to recreate a specific network behavior according to a defined model. The model adopted in this work is the *Gilbert Model* [8], one of the most-commonly applied amodel in performance evaluation studies, due to its analytical simplicity and the good results it provides in practical applications on wired IP networks [9]. The Gilbert Model is a 1-st order Markov chain model characterized by two states: state 0, with no

[14] www.planet-lab.eu
[15] www.cnrood.nl/PHP/files/telecom_pdf/Shunra_Virtual-Enterprise.pdf

losses, and state 1, with losses. There are four transition probabilities: (*i*) the probability to pass from state 0 to state 1 is called P, (*ii*) the probability to remain in state 0 is (1 - P), (*iii*) the probability to pass from state 1 to state 1 is called Q, and (*iv*) the probability to remain in state 1 is (1 - Q). Given PLR and ABL, P and Q are computed as follows: $P = \frac{PLR \cdot Q}{1-PLR}$ and $Q = ABL^{-1}$. The values for PLR and ABL have been obtained by a measurement campaign conducted on PlanetLab along some European Internet paths. Using these Measurements, we have defined two different scenarios: *Low Scenario*, with low values for PLR and ABL (respectively 1,35 and 0,59), and *High Scenario*, with higher values for PLR and ABL (respectively 5,05 and 1,44).

6.2 Results

First Experimental Campaign. In Figure 4C, the efficiency of the three best tree-based formats (*i.e.*, JSON, XML with SAX and compact YAML) is analyzed without compression and with each of the techniques described in Section 5. The highest efficiency, even better than the one of CDR, is achieved by using hybrid compression schemas. Figure 4A shows that bzip2 (*i.e.*, the techniques with best efficiency) is the one with worst performances, while the other two are quite similar. Figure 4B reveals that LZO achieves faster compression (and also decompression, not shown in the paper due to the limited available space), however, such strength is nullified by its lower compression efficiency, as shown in Figure 4C. Last, we can conclude that the best technique is zlib since it realizes the optimal trade-off between efficiency and performance.

Second Experimental Campaign. Figure 5 shows results with the testbed made with the Shunra emulator. Figure 5A proves that zlib is able to reduce the performance of tree-based formats both in Low and High Scenarios, behaving better than the other two hybrid schemas. However, there is still a strong difference than using CDR. However, another result shown in Figure 5B is surprising: data compression is a mean to increase the resiliency of the middleware. In fact, reducing the number of exchanged bytes can bring to less packets losses. Specifically, the technique with highest efficiency, *i.e.*, bzip2, presents the lower number of losses, achieving a resiliency degree close to CDR.

7 Conclusion

In the present article we have discussed the issue of providing flexible communication in pub/sub services. We argued that the commonly-used serialization formats, such as CDR and XML, are unsuitable since they do not exhibit an optimal trade-off between flexibility and performance. We have investigated the use of other two formats, such as JSON and YAML, which are claimed to be light-weighted version of XML. However, our results proved that, even using such formats, the performance overhead is still higher than the case of using the inflexible CDR. Then, we have proposed the introduction of data compression

to reduce such weakness. However, benefits in performance thanks to reducing the exchanged bytes is strongly mitigated by the time needed to perform compression and decompression operations. Last, we have conducted experiments in real case scenarios, which revealed us that data compression is also an effective solution to increase the resiliency by reducing the number of lost messages.

Acknowledgment

This work has been partially supported by the Italian Ministry for Education, University, and Research (MIUR) in the framework of the Project of National Research Interest (PRIN) "DOTS-LCCI: Dependable Off-The-Shelf based middleware systems for Large-scale Complex Critical Infrastructures".

References

1. Kolaitis, P.G.: Schema Mappings, Data Exchange, and Metadata Management. In: Proceedings of the ACM Symposium on Principles of Database Systems (PODS), pp. 90–101 (June 2005)
2. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.: The Many Faces of Publish/Subscribe. ACM Computing Surveys (CSUR) 35(2), 114–131 (2003)
3. Object Management Group, Data Distribution Service (DDS) for Real-Time Systems, v1.2, OMG Document (2007)
4. Object Management Group, Common Object Request Broker Architecture (CORBA), v3.0, OMG Document, pp. 15.4–15.30 (2002)
5. Zhao, J., Yang, D., Gao, J., Wang, T.: An XML Publish/Subscribe Algorithm Implemented by Relational Operators. In: Dong, G., Lin, X., Wang, W., Yang, Y., Yu, J.X. (eds.) APWeb/WAIM 2007. LNCS, vol. 4505, pp. 305–316. Springer, Heidelberg (2007)
6. Sayood, K.: Introduction to Data Compression, 3rd edn. Series in Multimedia Information and Systems. Morgan Kaufmann, San Francisco (2005)
7. Spring, N., Peterson, L., Bavier, A., Pai, V.: Using PlanetLab for network research: myths, realities, and best practices. ACM SIGOPS Operating Systems Review 40(1), 17–24 (2006)
8. Yu, X., Modestino, J.W., Tian, X.: The Accuracy of Gilbert Models in Predicting Packet-Loss Statistics for a Single-Multiplexer Network Model. In: Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '05), vol. 4, pp. 2602–2612 (March 2005)
9. Konrad, A., Zhao, B., Joseph, A.: Determining Model Accuracy of Network Traces. Journal of Computer and System Sciences 72(7), 1156–1171 (2006)

Mobile Agents for Digital Signage

Ichiro Satoh

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
ichiro@nii.ac.jp

Abstract. This paper presents an agent-based framework for building and operating context-aware multimedia content on digital signage in public/private spaces. It enables active and multimedia content to be composed from mobile agents, which can travel from computer to computer and provide multimedia content for advertising or user-assistant services to users. The framework automatically deploys their agents at computers near to their current positions to provide advertising or annotations on objects to users. To demonstrate the utility of the framework, we present a user-assistant that enables shopping with digital signage.

1 Introduction

Digital signage has been expected to play an important role in advertising, navigation, assistance, or entertainment in public and private environments, such as public museums, retail stores, and corporate buildings. Advertising using digital signage should be a form of out-of-home advertising (OOH) in which content and messages are displayed on digital signs with the common goal of delivering targeted messages to specific consumers in specific locations at specific times. For example, several retailers offer digital signage for sales promotions. The content that can be displayed on digital signage can be anything, including text, images, animations, video, audio, and interactivity. It has frequently been argued that digital signage needs to rely on quality content if it is to work effectively. Context-awareness is useful for digital signage. For example, the content for sales promotion on digital signage should be selected according to its target customers and displayed at terminals close to where they are currently located.

- Content, which is played on digital signage, can be anything, including text, images, animations, video, audio, and interactivity. We should provide a variety of multimedia content, including interactive content, to consumers, from digital signage.
- The content for sales promotion on digital signage should be selected according to its target items and customers and displayed at terminals that are close to the customers or items.
- Digital signage has its own spatial scope where people can see or listen to its content. User-specific content should be played only when its users are in its scope.
- Digital signage should be used for not only sales promotion but also to manage product life cycles, e.g., shipment, showcasing, assembly, usage, and disposal.

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 216–226, 2010.

– As context-aware digital signage should be provided in extensive building and city-wide areas, it cannot be managed by using traditional approaches, such as those that are centralized and top-down.

We constructed a general-purpose framework for managing context-aware annotation services [19]. The previous framework was designed for context-aware visitor assistance in museums. To support large-scale context-aware systems, it was managed in a non-centralized manner. It used mobile agent technology, where mobile agents were autonomous programs that could travel from computer to computer under their own control. Mobile agents were used as deployable services without any centralized management systems.

The framework presented in this paper is constructed based on the previous framework but it is designed for digital signage and can model the locations of people, products, and digital signage systems. This is because content displayed on digital signage needs to be selected and displayed according to people and physical entities. The framework needs to maintain a location model to manage digital signage. It introduces virtual counterpart objects for digital representations of people, physical entities, or digital signage systems, where each virtual counterpart object is a programmable entity. An application cannot directly interact with people or physical entities but with their virtual counterpart objects. Their counterpart objects interact with one another on behalf physical entities. The framework spatially binds the positions of people or entities with the locations of their virtual counterparts and, when they move from location to location in the real world, it automatically migrates their counterpart objects from the counterpart object corresponding to the source location to the counterpart object corresponding to the destination location.

Several researchers on virtual-reality (VR) have provided the notion of virtual scope, often called *aura*, where interactions between two objects in VR become possible only when the objects' scopes collide or overlap [4,7]. Digital signage can be seen by people when the former and latter are within a specified scope. For example, a public terminal has a half-meter sphere so that a user can directly manipulate the terminal. User-aware content should be displayed at digital signage only when the target user is in front of the digital signage. The framework introduces the scope of digital signage as a virtual counterpart. When a user is within the scope, his/her virtual counterpart is located in the virtual counterpart corresponding to the scope.

There have been several commercial projects for providing context-aware content on digital signage, but they have been constructed based in an ad-hoc manner. However, several researchers have explored context-aware services independently of the literature of digital signage. Cambridge University's Sentient Computing project [8] provided a platform for location-aware applications using infrared-based or ultrasonic-based locating systems in a building. Microsoft's EasyLiving project [3] enabled services running on different computers to be combined dynamically according to contextural changes in the real world. Here, we discuss differences between the framework presented in this paper and our previous frameworks. We constructed a location model for ubiquitous computing environments. The model represented spatial relationships between physical entities (and places) as containment relationships between their programmable counterpart objects and deployed counterpart objects at computers according to the positions

of their target objects or places [16]. This was a general-purpose location-model for context-aware services, but was not an infrastructure for deploying and operating such services. We presented an outline of mobile agent-based services in public museums in our early versions of this paper [19,20], whereas this paper addresses agent-based advertising on digital signage for shopping.

2 Design and Implementation

This section describes the current implementation of our framework.

2.1 Basic Approach

Like our previous framework [19], the framework introduces mobile agent technology. Content for digital signage is defined in mobile agents so that they can be dynamically deployed at computers close to users according to contexts in the real world, e.g., the locations of users and physical objects by using locating systems. Each mobile agent is a programmable entity with stored data. Therefore, each mobile agent-based services can define programs to play its visual/audio content and interact with users inside it. Therefore, the framework itself is independent of application-specific tasks and provides multiple kinds of multimedia content, because such tasks are performed within mobile agents.

Computers in digital signage only have limited resources, such as restricted levels of CPU power and amounts of memory. Mobile agents can help to conserve these limited resources, since each agent needs to be present at the computer only while the computer needs the content provided by that agent. After arriving at its destination, a mobile agent can continue work without losing the results of working, e.g., the content of instance variables in the agent's program, at the source computers. Therefore, users can continue to watch or listen to content from computers close to their current positions, even when the users move from location to location.

Existing location models can be classified into two types: physical-location and symbolic-location models [1,2,9]. The former represents the position of people and objects as geometric information. A few outdoor-applications like moving-map navigation can easily be constructed on the former. Most emerging applications, on the other hand, require a more symbolic notion, i.e., place, where place is the human-readable labeling of positions, e.g., the names of rooms and buildings. This paper addresses symbolic location as an event-driven programming model for context-aware digital signage. The model is maintained as a tree structure of virtual counterpart objects corresponding to people, physical entities, and digital signage according to containment relationships in the real world.

Digital signage has its own scope where people can see or listen to its content and it should be activated when people are with its scope. We introduce such a scope as virtual space, like the *aura* studied in virtual-reality research. Each digital signage can have a virtual counterpart object corresponding to the scope where people can see it in addition to the virtual counterpart corresponding to it. When a user is within the scope of a digital signage, his/her virtual counterpart object is located in the virtual

counterpart object corresponding to the scope and the latter activates the digital signage via the virtual counterpart object corresponding to the digital signage.

2.2 System Structure

The framework consists of three parts: (1) mobile agents, (2) agent runtime systems, (3) location model management systems (called LMMSs), and (4) location information servers (called LISs) (Fig. 1). The first offers application-specific content, which are attached to physical entities and places, as collections of mobile agents. The second runs on digital signage and is responsible for executing and migrating mobile agents. The third maintains the containment relationship of virtual counterparts to model the location of people or physical entities, e.g., products. The fourth provides a layer of indirection between the underlying location sensing systems and mobile agents. Each LIS manages more than one sensor and provides the agents with up-to-date information on the state of the real world, such as the locations of people, places, and things, and the destinations that the agents should migrate themselves to.

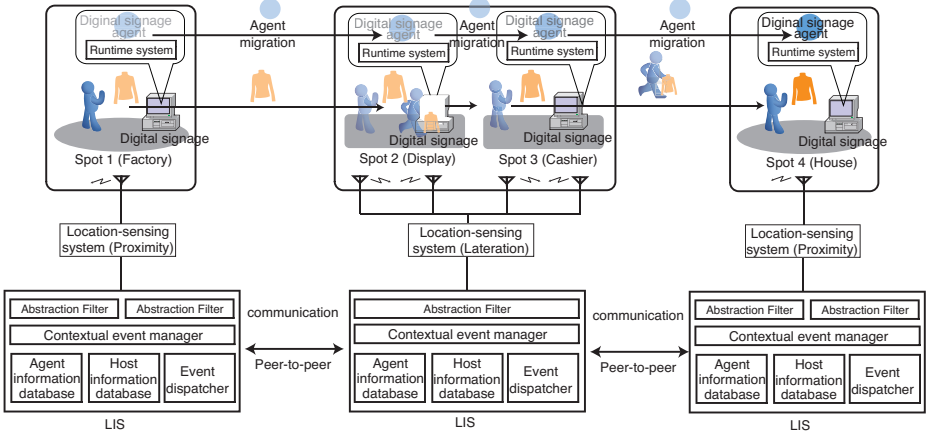


Fig. 1. Architecture

2.3 Location Information Server

The framework should be independent of its underlying location sensing systems. Location-sensing systems can be classified into two types: tracking and positioning systems. The former, including RFID tags, measures the location of other objects. The latter, including GPS, measures its own location. Since it is almost impossible to support all kinds of sensors, the model aims at supporting various kinds of tracking sensors, e.g., RFID-, infrared-, or ultra-sonic tags and computer vision, as much as possible. Each LIS can have a mechanism for managing location-sensors outside itself so that it is designed independently of sensors. It transforms geometric information about the positions of objects into corresponding containment relations. Each LIS is also responsible for discovering agents attached to people or physical entities. When an LIS detects

a new person or physical entity by using its sensing system, it sends a query message about agents attached to the person or entity to all LMMSs in its current sub-network by using UDP multicast communication. When an LMMS returns a reply message to the LIS, the LIS sends a control message to migrate the agents to a virtual counterpart corresponding to digital signage close to the current location of the new person or entity that is visting.

2.4 Agent Runtime System

The framework assumes that each digital signage supports a runtime system. Each runtime system is built on the Java virtual machine (Java VM) version 1.5 or later versions, which conceals differences between the platform architectures of the source and destination computers. It is responsible for executing agents. For example, it governs all the agents inside it and maintains the life-cycle state of each agent. When the life-cycle state of an agent changes, e.g., when it is created, terminates, or migrates to another runtime system, its current runtime system issues specific events to the agent.

Each runtime system running on digital signage has its own virtual counterpart agent (Fig. 2). The agent is located at a virtual counterpart corresponding to a space in a tree structure maintained by an LMMS according to the location of the signage in the real world. When it receives a mobile agent for content, it forwards the agent to its target digital signage and then the computer of the signage executes the visiting agent.

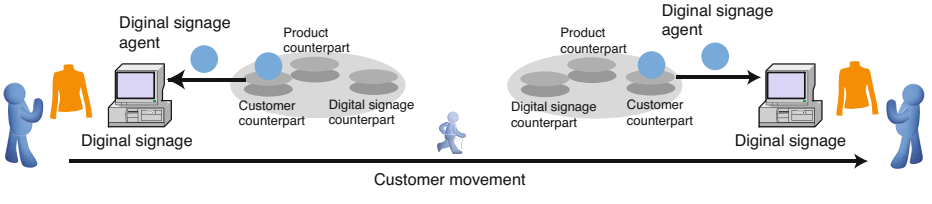


Fig. 2. Forwarding agents to digital signage when user moves

When an agent is transferred to digital signage over the network, not only the code of the agent but also its state is transformed into a bitstream by using Java's object serialization package and then the bit stream is transferred to the destination. Since the package does not support the capturing of stack frames of threads, when an agent is deployed at another computer, its runtime system propagates certain events to instruct it to stop its active threads. Arriving agents may explicitly have to acquire various resources, e.g., video and sound, or release previously acquired resources.

2.5 Location Model Management System

To model the locations of entities or places in the real world, we introduce a tree structure of virtual counterpart objects. Each counterpart object can be contained within at most one counterpart object according to containment relationships in the real world.

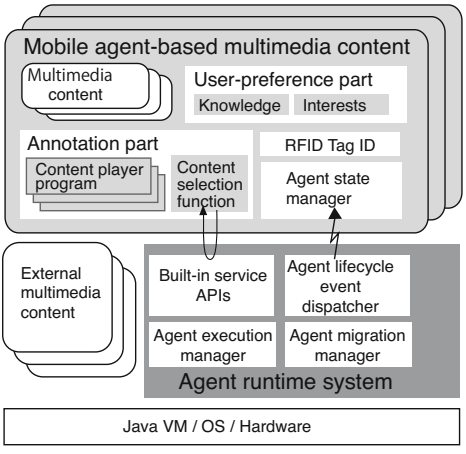


Fig. 3. Architecture of runtime system for service-provider agent

Each counterpart object can move between other counterpart objects. In the current implementation, each virtual counterpart object is constructed as a mobile agent with the notion of hierarchical mobile agents [14]. A mobile agent migrates to another agent, which may be running at a different computer, as a whole with all its inner agents. When physical entities move from location to location in the real world, an LIS detects their movements through location-sensing systems and changes the containment relationships of agents corresponding to moving entities, their source, and destination.

The tree structure of virtual counterpart objects can be maintained in more than one computer. Each runtime system enables counterpart objects to be organized in a subtree structure. We also introduce a link object, which is used as a proxy for a subtree that its target computer maintains and is located in the subtree that another computer maintains. As a result, it attaches the former subtree to the latter.

As mentioned previously, digital signage has its own scope where people can see or listen to its content. We introduce such a scope as virtual space, like *aura*. The framework allows each virtual space to define its shape and size. For example, a public terminal has a half-meter sphere so that a user can directly manipulate the terminal. However, there is a gap between the scope and the space where the underlying location sensing systems can detect the presence of the user. For example, such a scope may be smaller than the coverage areas of location sensing systems. Therefore, the current implementation introduces such a scope as the coverage area of a location sensing system.

2.6 Mobile Agent for Digital Signage Content

The content for digital signage and its virtual counterpart are implemented as mobile agents. Each mobile agent for providing content is attached to at most one product or user and defines programs that provide annotation and navigation to him/her (Fig. 3). To support user/location-dependent content, each agent is dynamically assembled from the *content* and *user-preference parts*.

Content part: This part is responsible for selecting and playing annotations according to users and products in addition to the information stored in the user-preference part and it plays the content in the personalized form of its user. It is defined as a set of a content-selection function and programs for playing the selected content.

The function maps more than one argument, e.g., the users and products into a URL referring to the annotative content. The content can be stored in the agent, the current runtime system, or external http servers. That is, each agent can carry a set of its content, play the selected content at its destinations, directly play the content stored at its destinations, or download and play the content stored in Web servers on the Internet. The current implementation can divide this part into three sub-parts: opening, annotation, and closing, which are played in turn.

Annotation content is varied, e.g., text, image, video, and sound. The annotation part defines programs for playing this content. The current implementation supports (rich) text data, html, image data, e.g, JPEG and GIF, video data, e.g., animation GIF and MPEG, and sound data, e.g., WAV and MP3. The format for content is specified in an MIME-based attribute description. Since the annotation part is defined as Java-based general-purpose programs, we can easily define interactions between visitors and agents.

User-preference part: This part is responsible for maintaining information about a visitor. In fact, it is almost impossible to accurately infer what a visitor knows or is interested in from data that have been measured by sensing systems. Instead, the current implementation assumes that administrators will explicitly ask visitors about their knowledge and interests and manually input the information into this part.

2.7 Current Status

This section describes the current implementation of our system. It was implemented using Sun's Java Developer Kit version 1.5 or later versions.

Support for location-sensing systems: The current implementation supports two commercial tracking systems. The first is the Spider active RFID tag system, which is a typical example of proximity-based tracking. It provides active RF-tags to users. Each tag has a unique identifier that periodically emits an RF-beacon (every second) that conveys an identifier within a range of 1-20 meters. The second system is the Aeroscout positioning system, which consists of four or more readers located in a room. These readers can measure differences in the arrival times of WiFi-based RF-pulses emitted from tags and estimate the positions of the tags from multiple measurements of the distance between the readers and tags; these measurement units correspond to about two meters.

Each LIS for the RFID tag system manages multiple sensors that detect the presence of tags and maintains up-to-date information on the identities of tags that are within the zone of coverage of its sensors. This is achieved by polling sensors or receiving the events issued by the sensors themselves. An LIS does not require any knowledge of other LISs. To conceal the differences among the underlying locating systems, each

LIS maps low-level positional information from each of the locating systems into information in a symbolic model of location. An LIS represents an entity's location, called a *spot*, e.g., a space of a few feet, which distinguishes one or more portions of a room or building. When an LIS detects a new tag in a spot, it multicasts a query that contains the identity of the new tag and its own network address to all the agent runtime systems in its current sub-network to discover agents tied to the tag. When there are multiple candidate destinations, each of the agents that is tied to a tag can select one destination on the basis of the profiles of the destinations. When the absence of a tag is detected in a spot, each LIS multicasts a message with the identifier of the tag and the identifier of the spot to all runtime systems in its current sub-network.

Security and privacy: The framework only maintains per-user profile information within those agents that are bound to the user. It promotes the movement of such agents to appropriate hosts near the user in response to the user's movements. Thus, the agents do not leak profile information on their users to other parties and they can interact with their mobile users in personalized form that has been adapted to respective, individual users. The runtime system can encrypt agents to be encrypted before migrating them over a network and then decrypt them after they arrive at their destination. Moreover, since each mobile agent is just a programmable entity, it can explicitly encrypt its particular fields and migrate itself with these fields and its own cryptographic procedure. The Java virtual machine can explicitly restrict agents to only access specified resources to protect hosts from malicious agents. Although the current implementation cannot protect agents from malicious hosts, the runtime system supports some authentication mechanisms for agent migration so that each agent host can only send agents to and only receive them from trusted hosts.

Performance evaluation: Although the current implementation was not built for performance, we measured the cost of migrating a null agent (a 5-KB agent, zip-compressed) and an annotation agent (1.2-MB agent, zip-compressed) from a source host to a destination host that was recommended by the LISs. The latency in discovering and instructing an agent attached to a tag after the CDS had detected the presence of the tag was 420 ms and the respective cost of migrating the null and annotation agent between the two hosts over a TCP connection was 38 ms and 480 ms. This evaluation was operated with three computers (Intel Core 2 Duo 2 GHz with Windows XP Professional and JDK 1.5) connected via a Fast Ethernet. This cost is reasonable for migrating agents between computers to that follow visitors moving between exhibits.

3 Early Experience

We experimented and evaluated mobile agent-based active content for appliances, e.g., electric lights. This was unique among other existing active content because it did not support advertising for its target appliance but assisted users to control and dispose of the appliance. We attached an RFID tag to an electric light and provided a mobile agent as an active content for the light. The content was attached to its target item and was deployed at computers close to the current position of the item. An agent for managing

active content on its target appliance is created when the appliance was shipped from its factory.

Since the agent defines programs to display three kinds of active content content inside it, it selects them according to their spaces. It supports the lifecycle of the item from shipment, showcasing, assembly, usage, and disposal.

- **In warehouse:** While the light is in a warehouse, its virtual counterpart agent is deployed at digital signage in the warehouse. It notifies a server in the warehouse of its specification, e.g., its product number, serial number, the date of its manufacture, and its size and weight.
- **In store:** While the light is being showcased in a store, its counterpart agent is deployed at a computer close to its target object to display advertising content to encourage customers who are visiting the store to buy it. Figure 4 a) and b) are two images maintained in the agent and they display the price, product number, and manufacture's name on its current computer.
- **In house:** After the light has been bought and taken to the house of its new owner, its agent migrates to digital signage in the house and illustrates how to assemble it. Figure 4 c) is the active content for the assembly manual. The agent also illustrates how to use it, as shown in Figure 4 d). When it is disposed of, the agent shows its active content to assist disposal guide. Figure 4 e) illustrates how to dispose of the appliance.

Fig. 4. Digital signage for supporting appliance

In a house-setting, we can define agents that control appliances, which may not have any network interfaces. In both of the approaches we have described here, the lights are controlled by switching their power sources on or off through a commercial protocol, called X10.

The first can autonomously turn room lights on with a tagged user is sufficiently close to them. The agent attached to the light can also work as our X10-based server's client and runs on the stationary runtime system in the room. When a tagged user approaches a light, an LIS in the room detects the presence of his/her tag in the cell that contains the light. The LIS then moves the agent that is bound to his/her tag to the runtime system

on which the light's agent is running. The user's agent then requests the lights' agent to turn the light on through inter-agent communication.

The second allows us to use a PDA to remotely control nearby lights. In this system, place-bound controller agents, which can communicate with X10-base servers to switch lights on or off, are attached to places with room lights. Each user has a tagged PDA, which supports the runtime system with WindowsCE and a wireless LAN interface. When a user with a PDA visits a cell that contains a light, the framework moves a controller agent to the runtime system of the visiting PDA. The agent, now running on the PDA, displays a graphical user interface to control the light. When the user leaves that location, the agent automatically closes its user interface and returns to its home runtime system.

4 Conclusion

We designed and implemented a context-aware infrastructure for building and managing mobile agent-based content displayed on digital signage, where mobile agents are autonomous programs that can travel from computer to computer under their own control as virtual counterpart objects for people or physical entities. It provides users and physical entities with mobile agent-based content to support and annotate them. Using location-tracking systems, it can migrate content to stationary or mobile computers near the locations of users and physical entities to which the agents are attached. That is, it allows a mobile user to access its personalized services in an active computing environment and provides user/location-dependent active content to a user's portable computer or stationary computer. It is managed in a decentralized manner. In addition, the system is managed in a non-centralized manner to support large-scale context-aware systems. Using the system, we constructed and operated two applications of location/user-aware multimedia on digital signage as case studies in our development of ambient computing services in public spaces.

Acknowledgement. This research is supported by Promotion program for Reducing global Environmental loaD through ICT innovation (PREDICT), Ministry of Internal Affairs and Communications of Japan.

References

1. Becker, C.: Context-Aware Computing. In: Tutorial Text in IEEE International Conference on Mobile Data Management (MDM 2004) (Januray 2004)
2. Beigl, M., Zimmer, T., Decker, C.: A Location Model for Communicating and Processing of Context. Personal and Ubiquitous Computing 6(5-6), 341–357 (2002)
3. Brumitt, B.L., Meyers, B., Krumm, J., Kern, A., Shafer, S.: EasyLiving: Technologies for Intelligent Environments. In: Thomas, P., Gellersen, H.-W. (eds.) HUC 2000. LNCS, vol. 1927, pp. 12–27. Springer, Heidelberg (2000)
4. Carlsson, C., Hagmnd, O.: DIVE: A platform for multi-user virtual environments. Computer and Graphics 17(6), 663–669 (1993)

5. Ciavarella, C., Paterno, F.: The Design of a Handheld, Location-aware Guide for Indoor Environments. Personal and Ubiquitous Computing 8(2), 82–91 (2004)
6. Fleck, M., Frid, M., Kindberg, T., Rajani, R., O'BrienStrain, E., Spasojevic, M., Spasojevic, M.: From Informing to Remembering: Deploying a Ubiquitous System in an Interactive Science Museum. IEEE Pervasive Computing 1(2), 13–21 (2002)
7. Greenhalgh, C., Benford, S.: MASSIVE: A Collaborative Virtual Environment for Teleconferencing. ACM Transactions on Computer-Human Interaction 2(3) (September 1995)
8. Harter, A., Hopper, A., Steggeles, P., Ward, A., Webster, P.: The Anatomy of a Context-Aware Application. In: Proceedings of Conference on Mobile Computing and Networking (MOBICOM'99), pp. 59–68. ACM Press, New York (August 1999)
9. Leonhardt, U., Magee, J.: Towards a General Location Service for Mobile Environments. In: Proceedings of IEEE Workshop on Services in Distributed and Networked Environments, pp. 43–50. IEEE Computer Society, Los Alamitos (1996)
10. Luyten, K., Coninx, K.: ImogI: Take Control over a Context-Aware Electronic Mobile Guide for Museums. In: Workshop on HCI in Mobile Guides, in conjunction with 6th International Conference on Human Computer Interaction with Mobile Devices and Services (2004)
11. Oppermann, R., Specht, M.: A Context-Sensitive Nomadic Exhibition Guide. In: Thomas, P., Gellersen, H.-W. (eds.) HUC 2000. LNCS, vol. 1927, pp. 127–142. Springer, Heidelberg (2000)
12. Richardson, T., Stafford-Fraser, Q., Wood, K., Hopper, A.: Virtual Network Computing. IEEE Internet Computing 2(1), 33–38 (1999)
13. Rocchi, C., Stock, O., Zancanaro, M., Kruppa, M., Kruger, A.: The Museum Visit: Generating Seamless Personalized Presentations on Multiple Devices. In: Proceedings of 9th International Conference on Intelligent User Interface, pp. 316–318. ACM Press, New York (2004)
14. Satoh, I.: MobileSpaces: A Framework for Building Adaptive Distributed Applications Using a Hierarchical Mobile Agent System. In: Proceedings of International Conference on Distributed Computing Systems (ICDCS 2000), pp. 161–168. IEEE Computer Society, Los Alamitos (2000)
15. Satoh, I.: SpatialAgents: Integrating User Mobility and Program Mobility in Ubiquitous Computing Environments. Wireless Communications and Mobile Computing 3(4), 411–423 (2003)
16. Satoh, I.: A Location Model for Pervasive Computing Environments. In: Proceedings of IEEE 3rd International Conference on Pervasive Computing and Communications (PerCom'05), pp. 215–224. IEEE Computer Society, Los Alamitos (March 2005)
17. Satoh, I.: Building and Selecting Mobile Agents for Network Management. Journal of Network and Systems Management 14(1), 147–169 (2006)
18. Satoh, I.: A Location Model for Smart Environment. Pervasive and Mobile Computing 3(2), 158–179 (2007)
19. Satoh, I.: Context-aware Deployment of Services in Public Spaces. In: Brinkschulte, U., Givargis, T., Russo, S. (eds.) SEUS 2008. LNCS, vol. 5287, pp. 221–232. Springer, Heidelberg (2008)
20. Satoh, I.: A Context-aware Service Framework for Large-Scale Ambient Computing Environments. In: Proceedings of ACM International Conference on Pervasive Services (ICPS'09), pp. 199–208. ACM Press, New York (July 2009)

Composition Kernel: A Multi-core Processor Virtualization Layer for Rich Functional Smart Products

Tatsuo Nakajima, Yuki Kinebuchi, Alexandre Courbot, Hiromasa Shimada, Tsung-Han Lin, and Hitoshi Mitake

Department of Computer Science and Engineering
Waseda University
tatsuo@dcl.info.waseda.ac.jp

Abstract. Future ambient intelligence environments will embed powerful multi-core processors to compose various functionalities into a smaller number of hardware components. This makes the maintainability of intelligent environments better because it is not easy to manage massively distributed processors.

A composition kernel makes it possible to compose multiple functionalities on a multi-core processor with the minimum modification of OS kernels and applications. A multi-core processor is a good candidate to compose various software developed independently for dedicated processors into one multi-core processor to reduce both the hardware and development cost. In this paper, we present SPUMONE which is a composition kernel for developing future smart products.

1 Introduction

Multi-core processors are being increasingly adopted for embedded systems because they improve performance, power consumption and lower development cost. Composing multiple operating systems on a multi-core processor enhances the reusability of software when developing rich functional embedded systems. For example, a new product may require to use the new version of an operating system, but to ensure the compatibility with legacy software, the old version may also need to be present. Multiple OS environments enable the product to use two versions of an operating system at the same time. In order to build multiple OS environments, a virtualization layer specialized for embedded systems is necessary, since most of processors for embedded systems support only two protection levels, and there is no hardware support for virtualization. In traditional approaches, an OS kernel runs at the user level to isolate the respective OS kernels, but this approach requires heavy modifications to the guest OSes. Especially, device drivers need to be rewritten and may degrade the performance significantly. Therefore existing solutions are not preferred by the embedded system industry.

In this paper, we propose a composition kernel where multiple OS kernels are running on top of a very thin hardware abstraction layer. The hardware abstraction layer multiplexes underlying physical processor cores into virtual cores

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 227–238, 2010.

which can be dynamically migrated among the physical cores. A composition kernel can reduce the engineering cost of developing an embedded system by reusing existing OS kernels and application with minimum modification. It also supports real-time interrupt responsiveness, a feature that is difficult to support for large and highly functional monolithic OS, like Linux and Windows. In addition, flexible virtual core migration can help reducing the power consumption of the processor.

Our project is developing SPUMONE which is a composition kernel for embedded systems, and currently focuses on the following three issues.

- Mapping and scheduling of virtual cores on physical cores dynamically to balance the tradeoff among real-time constraints, performance and energy consumption.
- Reducing interrupt latency without degrading real-time performance in a single and multi-core processor.
- Detecting the integrity violations in OS kernels, and repairing them by rebooting the kernels independently.

SPUMONE offers a scheduling algorithm to execute a general purpose OS without affecting the timing constraints of real-time OSes. Also, the execution of general purpose OSes should utilize the maximum performance of multi-core processors. However, when the system load becomes low, SPUMONE reduces the number of used physical cores by migrating virtual cores to a small number of physical cores. The unused physical cores can be turned off to reduce the power comsumption. The overview of multi-core resource management is described in Section 4.1.

For satisfying timing constraints of real-time OSes, SPUMONE carefully coordinates interrupt handling to reduce the effect of disabling interrupts in a single processor case. However, in a multi-core processor case, SPUMONE migrates a virtual core that executes a general purpose OS on another physical core when a real-time OS becomes runnable. Also, lock holder preemption is a serious problem to run an SMP OS kernel on SPUMONE. The migration of virtual cores can solve the problem as described in Section 4.2.

A monitoring service increases the security and reliability of the entire system. On SPUMONE, each OS kernel can be rebooted independently when the kernel is crashed. The monitoring service enables a kernel to be rebooted proactively. When the monitoring service detects an anomaly in the kernel by checking the integrity of some of its data structures, it tries to restore the integrity of these data structures. If the anomaly cannot be fixed, the monitoring service will reboot the OS kernel to recover its integrity completely. This approach can be used to remove kernel rootkits that modify the behavior of operating system kernels. The overview of reliability and security issues in SPUMONE is described in Section 4.3.

2 Motivation

In the near future, a variety of daily objects near us will become smart products. These artifacts are connected to the Internet and enhance our daily activities. In our research group, we have enhanced various daily objects such as chairs, tables, toothbrushes, and mirrors [12]. These products have the surfaces to encourage people to motivate desirable behavior [6,7], or provide the economic incentives [10]. This offers us a big opportunity to make traditional products more attractive.

There are two characteristics to develop these future smart products. The first is to offer a huge amount of functionalities that need to satisfy diverse requirements to offer various attractive services. These diverse requirements cannot be implemented on only one operating system. Current smart products adopt various types of operating systems to satisfy different requirements. For example, products controlling a variety of devices have used small operating systems that include only a real-time thread scheduler and some device drivers. The operating systems usually do not support memory protection domains, but are suitable for implementing highly responsive services with tight timing constraints.

Diverse hardware platforms are the second characteristic. Especially, future smart products will need to use a multi-core processor dynamically to save energy consumption. As described in the previous paragraph, multiple operating systems should be executed on a multi-core processor. Each operating system allocates a suitable number of CPU cores according to the current workload. Let us assume a mobile phone that uses a multi-core processor. While a user does not use the mobile phone, only one CPU core is used to execute several background application services on multiple operating systems simultaneously. In this case, it is easy to satisfy all real-time requirements of these activities on a single CPU core by migrating all operating systems on the core. However, when a user starts watching a TV program, multiple CPU cores become active and most of them are used to process the TV program. The mapping the execution of operating systems and physical CPU cores should be flexible according to the current workload.

Dependability is one of the most important requirements in future smart products. Crashing or hanging of a service on an appliance will degrade user experience significantly. For example, if the service is hung, a user needs to find a reset button and push it to restart the appliance. Usually, a user interacts with information appliances for a short time. Although some errors inside a kernel may damage the kernel, the appliance can usually be avoided to crash while a user is interacting with it by repairing a small amount of damaged kernel's data structure. The kernel will be restarted for achieving a complete repair after a user stops to use the appliance.

3 Composition Kernel: SPUMONE

Multi-core processors will become more and more common for future information appliances. Composing multiple OS functionalities is a promising approach to use

multi-core processors effectively. As the number of functionalities increases, the system requires more computation power to execute them without violating their performance requirements. Since different functionalities are often implemented on different operating systems, an underlying software platform needs to execute multiple operating systems without having to reimplement them on one single operating system. This approach enables a sytem to reuse existing application and operating system code. Thus, it allows to develop smart products with rich functionalities easily and at a low cost.

There are several traditional approaches to execute multiple operating systems on a processor in order to compose multiple functionalities. Microkernels execute guest operating system kernels at the user level. In this case, various privileded instructions, traps and interrupts need to be virtualized by replacing their code. Also, since operating system kernels are to be executed as user level tasks, application tasks need to communicate with the operating system kernel via inter-process communication. Moreover, when emulating Unix-based operating systems, implemeting signals using this approach is very hard. Therefore, the operating system needs to be modified to a significant amount. Virtual machine monitors are another approach to execute multiple operating systems. If a processor offers a hardware virtualization support, all instructions that need to be virtualized trigger traps to the virtual machine monitor. This makes it possible to use the operating system without any modification. But, if the hardware virtualization support is incomplete, some instructions still need to be complemented by replacing some codes to virtualize them.

Most of processors used for embedded systems only have two protection levels, and MMU cannot usually be used in the kernel address space. So, when operating system kernels are located in the kernel address space, they are hard to be isolated. On the other hand, if the operating system kernels are located in the user address space, the kernels need to be modified siginificantly. Most of embedded system industries prefer not to modify a large amont of the operating systems' code, so it is desirable to put them in the kernel address space. Also, the virtualization of MMU requires significant overhead if the virtualization is implemented by software[1]. Therefore, we need altenative mechanisms to ensure the security and reliability of the kernels.

In a traditional virtual machine monitor, handling I/O devices causes a significant problem. When isolating device drivers from operating systems, the engineering cost is very serious. I/O ports can be emulated by virtual machine monitors, but such an approach causes serious performance degradation. If microkernels are used, device drivers can be implemented in user-level OS kernels. In this case, it is possible to allow the OS kernels to access kernel memory through DMA. Although device drivers can be separated from the OS kernels, the drivers need to be re-implemented with a significant engineering cost. Finally, device drivers can be implemented inside virtual machine monitors or microkernels, but this approach decreases the reliability and security due to potential bugs in device drivers.

[1] Xen incurs 30% overhead if MMU is virtualized using the shadow paging.

In order to execute multiple operating system kernels on a multi-core processor, the assignment of OS kernels to physical cores should be taken into account. The underlying platform offers virtual cores to OS kernels. Virtual cores are used to schedule all activities in the OS kernels, and the mapping between physical cores and virtual cores should be completely transparent to OS kernels without increasing engineering cost.

In [11], Armand and Gien present several requirements for hardware virtualization for embedded systems:

- Run an existing operating system and its supported applications in a virtualized environment, such that modifications required to the operating system are minimized (ideally none), and performance overhead is as low as possible.
- It should be straightforward to move from one version of an operating system to another one; this is especially important to keep up with frequent Linux evolutions.
- Reuse native device drivers from their existing execution environments with no modifications.
- Support existing legacy often real-time operating systems and their applications while guaranteeing their deterministic real-time behavior.

In our project, we are developing a composition kernel called SPUMONE. SPUMONE (Software Processing Unit, Multiplexing ONE into two or more) is a thin software layer for multiplexing a single physical CPU core into multiple virtual ones. In this section, several characteristics are shown as follows.

Virtualization Strategies

Unlike typical microkernels or virtual machine monitors, SPUMONE itself and OS kernels are executed in the privileged mode as mentioned in this section. Executing SPUMONE and OS kernels in the privileged mode contributes to minimize the overhead introduced to and the amount of modifications required to the OS kernels. Furthermore it makes the implementation of SPUMONE itself simple. Executing OS kernels in the user mode is known to complicate the implementation of the virtualization layer, because various privileged instructions need to be emulated. In our approach, the majority of the kernel and application instructions, including the privileged instructions, are executed directly by the real CPU core, and only a minimal set of instructions are emulated by SPUMONE. These emulated instructions are invoked from the OS kernels using simple function calls. Since the interface has no binary compatibility with the original CPU core interface, we simply modify the source code of OS kernels, a method known as the paravirtualization.

For isolating multiple operating systems, if it is necessary, SPUMONE assumes that underlying processors support the mechanisms to protect physical memories used by respective operating systems like VIRTUS [4]. The approach may be suitable for enhancing the reliability of the OS kernels on SPUMONE without increasing significant overhead.

SPUMONE does not virtualize IO devices because traditional approaches incur significant overhead that most of embedded systems could not tolerate. In

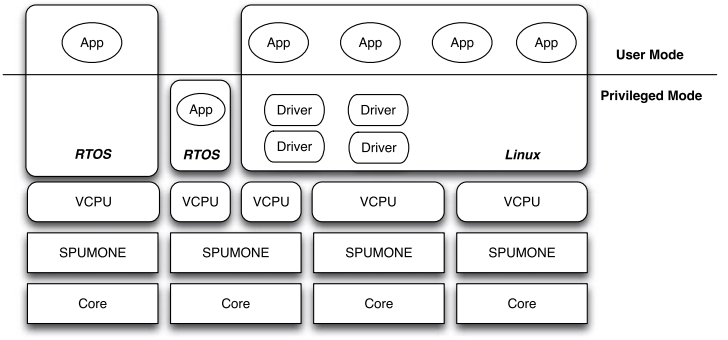


Fig. 1. Composition Kernel

SPUMONE, since device drivers are implemented in the kernel address space, they do not need to be modified when the device is not shared by multiple operating systems.

Interrupt/Trap Delivery
Interrupt virtualization is a key feature of SPUMONE. Interrupts are investigated by SPUMONE before they are delivered to each OS. SPUMONE receives an interrupt, then looks up the interrupt destination table to make a decision to which OS it should be delivered. The destination virtual core is statically defined for each interrupt when the kernels are built. Traps are also delivered to SPUMONE first, then are directly forwarded to the currently executing virtual core.

The interrupt delivery process on a multi-core platform works basically like the one on a single-core platform. Each SPUMONE instance delivers interrupts to their destinations. In order to deliver interrupts to a virtual core running on a different core, the assignments of interrupts and physical cores are switched along with virtual core migrations.

Virtual Core Scheduling
A CPU core is multiplexed by scheduling the execution of virtual cores. The execution states of OSes are managed by a data structure that we call a *vcpu*. When switching the execution of virtual cores, all the hardware registers are stored into the corresponding vcpu's register table, and then loaded from the table of the next executing vcpu. The mechanism is similar to the process implementation of a classical OS, but in addition, SPUMONE saves the entire processor state, including the privileged control registers.

The scheduling algorithm of virtual cores is the fixed priority preemptive scheduling. When the real-time OS and the general purpose OS share a physical core, the virtual core bound to the RTOS would gain a higher priority than the virtual core bound to the general purpose OS in order to maintain the real-time responsiveness. This means the general purpose OS is executed only when the

virtual core for the real-time OS is in an idle state and has no task to execute. The process or task scheduling is left up to OS so the scheduling model for each OS is maintained as-is. The idle real-time OS resumes its execution when it receives an interrupt. When virtual cores assigned to the general purpose OS are migrated to execute on a shared core, those cores are scheduled with a timesharing scheduler.

4 Highlights in SPUMONE

In the current implementation, we adopted Toppers[2] as a real-time OS and Linux as a general purpose OS. Also, we modified SMP Linux to use multiple virtual core offered by SPUMONE to exploit multi-core processors.

We have implemented SPUMONE on the Hitach/Renesas RP1 experimental multi-core board. The processor contains four SH4-A cores which can communicate with a shared memory. Currently, Linux, Toppers and OKL4 are running on SPUMONE. The modification of guest OSes is usually less than about 100 lines. The worst case interrupt latency of Toppers is less than 35μs while executing Linux both on a single and multi-core processor.

4.1 Dynamic Multiple Cores Management

SPUMONE for multi-core processors is designed in a distributed model. A dedicated instance of SPUMONE is assigned to each physical core as shown in Fig.1. This design is chosen in order to eliminate the unpredictable overhead of synchronization among multiple CPU cores. In addition, the basic lock mechanism can be shared between single-core and multi-core version, which may simplify the design of SPUMONE. It also enables the system to scale on multi-core and many-core processors as discussed in [1].

As described in the previous section, SPUMONE enables to multiplex multiple virtual cores on physical cores. The mapping between physical cores and virtual cores is dynamically changed to balance the tradeoffs among real-time constraints, performance and energy consumption. In SPUMONE, a virtual core can be migrated to another core according to the current situation. There are several advantages of our approach.

The first advantage is to change the mapping between virtual cores and physical cores to reduce energy consumption. As shown in Fig. 2, we assume that a processor offers two physical cores. Linux uses two virtual cores, and the real-time OS uses one virtual core. When the utilization of Toppers is high, two virtual cores of Linux are mapped on one physical core (Left Top). When Toppers is stopped, each virtual core of Linux uses a different physical core (Right Top). Also, one physical core is used by a virtual core of Linux and another physical core is shared by Linux and Toppers when the utilization of Toppers

[2] Toppers is an open source real-time OS used in various Japanese embedded system products. Toppers implements the μITRON interface specification that is a Japanese standard for the real-time OS.

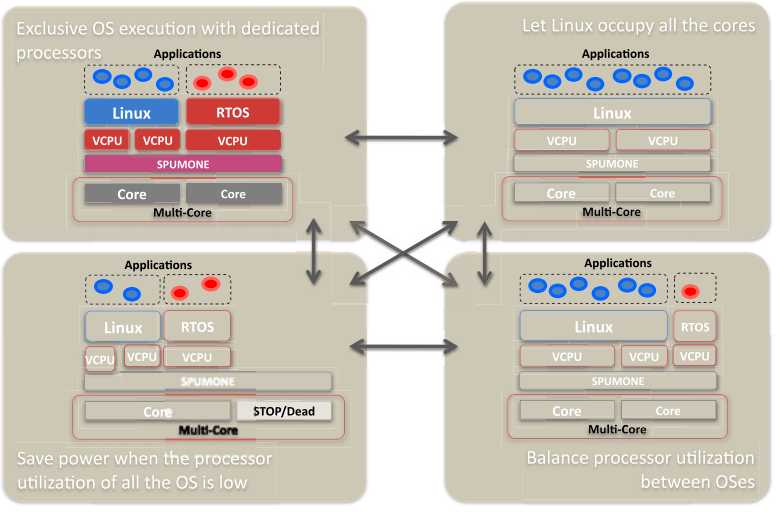


Fig. 2. Dynamic Multiple Cores Management

is low (Right Below). Finally, when it is necessary to reduce energy consumption or one of physical cores is dead, all virtual cores run on one physical core (Left Below). This approach enables us to use very aggressive policies to balance real-time constraints, performance, and energy consumption.

4.2 Reducing Interrupt Latency

In order to minimize interrupt delay of Toppers while sharing a physical core by Linux and Toppers, we proposed two approaches for a single and multi-core processor respectively.

The first approach that is for a single core processor is replacing the interrupt enable and disable instructions with the virtual instruction interface. A typical OS disables all interrupt sources when disabling interrupts for atomic execution. Our approach leverages the interrupt mechanism of the processor: we assign the higher half of the interrupt priority levels (IPLs) to Toppers and the lower half to Linux (Fig.3: Left). The instructions enabling and disabling interrupts are typically provided as kernel internal APIs. They are typically coded as inline functions or macros in the kernel source code.

When the Linux tries to block the interrupts, it modifies its interrupt mask to the middle priority. Toppers may therefore preempt Linux even if it is disabling the interrupts (Fig.3: Right (1)). On the other hand when Toppers is running, the interrupts are blocked by the processor (Fig.3 : Right (2)). These blocked interrupts could be immediately delivered when Linux is dispatched.

The second approach that is for a multi-core processor is based on virtual core migration. As we implemented the first approach described in the previous paragraph, we found that some paths in the Linux kernel gained a highest lock

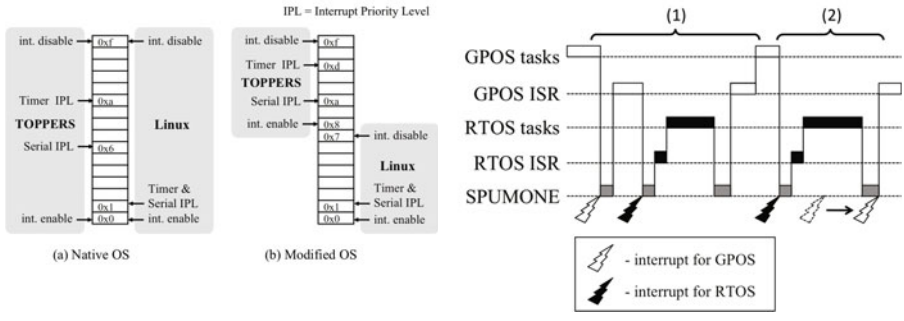


Fig. 3. The interrupt priority levels assignment and Interrupt Delivery Mechanism

priority unexpectedly (e.g. bootstrap, idle thread). This suggests us the possibility that some device drivers or kernel modules programmed in a bad manner gain a high IPL and interfere with the activity in Toppers. This means that careful coordination of IPLs requires high engineering cost. We modified SPUMONE to proactively migrate a virtual core, which is assigned to Linux that shares a physical core with Toppers, to another physical core when it traps into the Linux kernel or interrupts are triggered. In this way, only the user level code of Linux is executed concurrently on a shared physical core, which will never modify the IPLs. Therefore, Toppers may preempt Linux immediately without separating IPLs used in the first approach (Fig.4).

The approach can also minimize the effect of lock holder preemption. When the lock holder in the Linux kernel is preempted by Toppers, the Linux kernel executed on other physical cores must wait until Toppers becomes idle, and the lock owned by the preempted Linux kernel is released. This significantly degrades the performance of Linux. The virtual core migration ensures that the execution of the Linux kernel is always migrated to other physical cores that do not execute Toppers.

4.3 Security and Reliability

In SPUMONE, guest OS kernels share the same priviledged address space to reduce the amount of modifications and performance impact as much as possible. As a consequence, we need another approach to enhance security and reliability without relying on familiar isolation mechanisms. The basic approach is to use a monitoring service to detect the violation of the integrity of each kernel, and recover by rebooting the kernel. In our approach, each guest OS kernel can be rebooted independently even though all OS kernels reside in the same address space.

The monitoring service checks the integrity of several data structures in the OS kernel periodically. The integrity is specified as constraints of each data

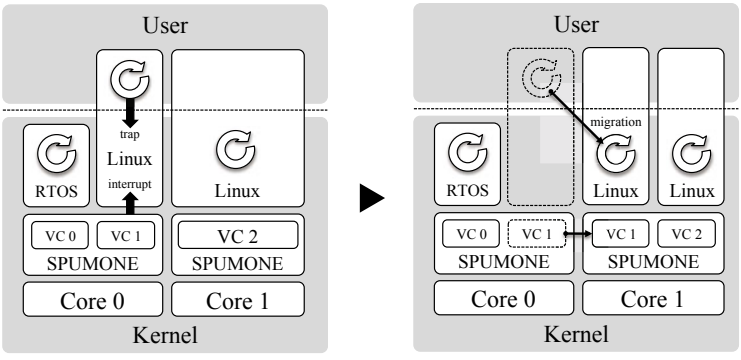


Fig. 4. Virtual core migration

structures. If the monitoring service detects the violation of the constraints, the service invokes a recovery function that is defined for each data structure to recover the integrity. The repair procedure may not repair the system completely - some garbage may remain in the kernel space or the repair procedure may even fail. In this case, the guest OS kernel is rebooted proactively.

When the Linux kernel causes an error while executing the kernel, the error can be translated to a system call error or an application signal. Of course, the optimistic approach may leak some resources in the kernel. In this case, Linux is rebooted when the kernel becomes idle. This approach is very effective in some embedded systems. For example, when the user is using a mobile phone, the Linux kernel does not need to be rebooted immediatly if some errors occur in the kernel, but the kernel can be rebooted when the user puts the phone in his pocket.

We are also considering an alternative approach. When the monitoring service detects some anomalies, it saves the states of application processes. Then, the Linux kernel is rebooted, and the states of processes are reconstructed. The applications can continue to run even though the kernel is restarted, which is similar to the checkpoint/recovery approach. Toppers and its applications can be simply rebooted when some anomalies are detected. The rebooting time can be improved by storing some important states in a persistent memory by using a similar techniques presented in [5]. Usually, most applications on Toppers contain a small amount of states, and rebooting the applications and Toppers is very fast. Also, the rebooting does not affect the functionality of the embedded system. This approach improves user satisfaction dramatically because the user is not aware of the reboot.

In our approach, if the Linux kernel is attacked, the attacker can invade other OS kernels. In order to attack other kernels, an attacker needs to insert code into the Linux kernel address space. Various traditional approaches can detect the modifications of the kernel easily. Recently, attacks tend to use kernel rootkits. Kernel rootkits try to stealth themselves, and various security tool cannot find them. For example, some rootkits may modify kernel data structures that are

used to manage processes. In our approach, the monitoring service checks the data structures that the rootkits try to modify, and repairs them to allow security tools to detect the rootkits.

Currently, the monitoring service checks some typical data structures that various rootkits are known to modify, and shows that our approach can remove many well known rootkits. We are also working on the synchronization mechanism between the monitoring service and the Linux kernel. Our approach uses optimistic synchronization because we cannot modify the Linux kernel to exclude shared data structures between the Linux kernel and the monitoring service [9].

Of course, the monitoring service should be protected from the Linux kernel. There are various mechanisms to protect the monitoring service. For example, we can use a co-processor or a special device to execute the monitoring service. The multi-core processor that we are using (SH4-A) for building embedded systems contains a local memory in each CPU core. This local memory can only be accessed by its CPU core. In our approach, a CPU core is dedicated to execute the monitoring service. Thus, the Linux kernel cannot access its local memory, but the CPU core executing the monitoring service can access all the memory used by the Linux kernel.

5 Current Status and Future Direction

SPUMONE can execute multiple operating systems without suffering a large amount of overhead and engineering cost. Although most of processors for embedded systems are not suitable to implement the virtualization layer to offer the complete isolation between guest OSes because it requires a large amount of overhead without virtualization hardware supports. Since SPUMONE and OS kernels run in the same privileged space, our approach increases the possibility of the kernel corruption, but a monitoring service detects the corruption in SPUMONE and guest OSes and heals them by rebooting. Also, introducing the virtualization layer in embedded systems offers additional advantages. For example, proprietary device drivers can be mixed with GPL codes without license violation. This solves various business issues when adopting Linux in embedded systems.

We are currently enhancing our implementation to support various policies to consider the tradeoff among power consumption, performance and timing constraints. The monitoring systems should be enhanced in the near future. Especially, we are interested in using Daikon [2] to detect the invariance inside the kernel automatically.

Asymmetric multicore processors are a promising approach to reduce the power consumption [3]. In SPUMONE, we are considering to hide the heterogeneity inside the SPUMONE and offer virtual homogeneous multicore processors to operating systems. However, it is not easy to hide the heterogeneity completely. We are also considering to develop the MapReduce-based applications on SMP Linux [8], and coordinate the middleware and SPUMONE to hide the heterogeneity completely.

References

1. Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schuepbach, A., Singhania, A.: The Multikernel: A New OS Architecture for Scalable Multicore Systems. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles (2009)
2. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon System for Dynamic Detection of Likely Invariants. Science of Computer Programming 69(1-3), 35–45 (2007)
3. Fedorova, A., Saez, J.C., Shelepov, D., Prieto, M.: Maximizing Power Efficiency with Asymmetric Multicore Systems. Communication of the ACM 52(12), 48–57 (2009)
4. Inoue, H., Sakai, J., Edahiro, M.: Processor virtualization for secure mobile terminals. ACM Transaction on Design Automation of Electronic Systems 13(3) (2008)
5. Ishikawa, H., Courbot, A., Nakajima, T.: A Framework for Self-Healing Device Drivers. In: Proceedings of the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, pp. 277–286 (2008)
6. Nakajima, T., Lehdonvirta, V., Tokunaga, E., Kimura, H.: Reflecting Human Behavior to Motivate Desirable Lifestyle. In: Proceedings of the Conference on Designing Interactive Systems, pp. 405–414 (2008)
7. Nakajima, T., Kimura, H., Yamabe, T., Lehdonvirta, V., Takayama, C., Shiraishi, M., Washio, Y.: Using Aesthetic and Empathetic Expressions to Motivate Desirable Lifestyle. In: Proceedings of the Third European Conference on Smart Sensing and Context, pp. 220–234 (2008)
8. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating MapReduce for Multi-core and Multiprocessor Systems. In: Proceedings of the 13th Intl. Symposium on (2007)
9. Shimada, H., Courbot, A., Kinebuchi, Y., Nakajima, T.: A Lightweight Monitoring Service for Multi-Core Embedded Systems. In: Proceedings of the 13th IEEE Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (2010)
10. Yamabe, T., Lehdonvirta, V., Ito, H., Soma, H., Kimura, H., Nakajima, T.: Applying Pervasive Technologies to Create Economic Incentives that Alter Consumer Behavior. In: Proceedings of the 11th International Conference on Ubiquitous Computing, pp. 175–184 (2009)
11. Armand, F., Gien, M.: A Practical Look at Micro-Kernels and Virtual Machine Monitors. In: Proceedings of the IEEE 6th Consumer Communications and Networking Conference 2009, pp. 1–7 (2009)
12. Kawsar, F., Nakajima, T., Fujinami, K.: Deploy Spontaneously: Supporting End-Users in Building and Enhancing a Smart Home. In: Proceedings of the 10th International Conference on Ubiquitous Computing, pp. 282–291 (2008)

Mobile Phone Assisted Cooperative On-Node Processing for Physical Activity Monitoring*

Robert Diemer and Samarjit Chakraborty

Institute for Real-time Computer Systems, TU Munich, Germany
diemer@tum.de, samarjit@tum.de

Abstract. One of the main challenges in the body-area sensor network domain is to suitably break complex signal processing tasks into manageable parts in order to reduce their algorithmic complexity while retaining their output quality. The goal is to map some of these tasks onto sensor nodes and the others onto computation platforms or gateways like mobile phones. In this paper we attempt to address this problem in the specific context of physical activity monitoring. To start with, physical activity *recognition* tasks are carried out on the mobile phone. But as soon as a steady-state (e.g., walking or running at constant speed) is detected, this information is transmitted to the sensor node. At this stage, the sensor node *monitors* the known physical activity, which entails relatively simpler algorithms. In the event of a change in activity pattern, it switches back to raw data transmission and hands over processing to the mobile phone. Such cooperative signal processing significantly improves the battery life of the mobile phone as well as that of the sensor node. We present the main principles behind such distributed physical activity monitoring algorithms and compare their output quality with those from standard processing done entirely on the mobile phone.

1 Introduction

Lately, there has been a tremendous amount of interest in augmenting personal portable devices like mobile phones with lifestyle or health-monitoring applications. Sensor nodes attached to the user's body or clothing continously transmit sensed information to the mobile phone, which is then processed to record various physical activity patterns or the health condition of the user (e.g., body temperature and pulse rate). In this paper we consider a setup with a mobile phone and a small sensor board with a triaxial accelerometer, which are connected via Bluetooth. Typically, because of the limited processing capability of the microcontroller on the sensor board, all data is transmitted to the mobile phone. Several signal processing algorithms are then run on the phone in order to carry out the necessary physical activity recognition tasks. However, such continuous data reception and processing is against the standard usage pattern of a mobile phone and hence stresses its processor excessively. As a result, it is

* This work was supported by German Federal Ministry of Education and Research (01FC08069).

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 239–251, 2010.

unable to go into deep power-down modes, thereby draining its battery quickly. In particular, our experiments show that the battery life of the mobile phone reduces significantly if it is constantly subjected to data reception from the sensor node and the associated processing tasks. Further, the continuous wireless data transmission severely shortens the battery life of the sensor node too.

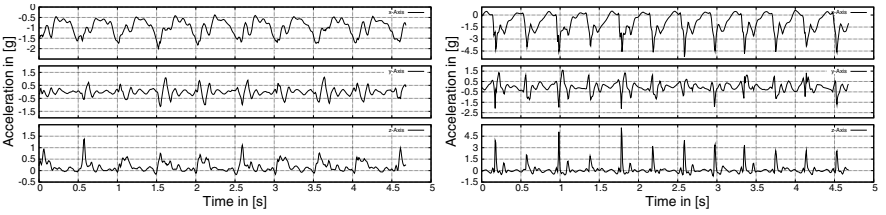
Our contributions: This is a standard problem, whose solution is to partition complex signal processing tasks and suitably map some of them onto the senor node and others onto the mobile phone with the aim of retaining activity recognition/monitoring quality and at the same time save power. In this paper we propose such a cooperative and distributed signal processing solution for physical activity recognition and monitoring. In particular, our algorithms process triaxial acceleration signals in order to classify and monitor the following activity types: level walking, walking up stairs, walking down stairs, stand-to-sit transition, and sit-to-stand transition. The first three activity types can be generally considered as *locomotive* in nature while the remaining two activity types are *postural transitions*. Besides these gait and physical activities, the general motionless resting state can also be identified using our algorithms. Since the computational capability of our sensor node [4] is limited, the physical activity *recognition* tasks are carried out on the mobile phone. But as soon as a steady-state (e.g., walking or running at constant speed) is detected, this information is transmitted to the sensor node. At this stage, the sensor node *monitors* the known physical activity, which entails relatively simpler algorithms. In the event of a change in activity pattern, it switches back to raw data transmission and hands over processing to the mobile phone. The information gathered during the monitoring phase can either be used to compute metrics like the amount of energy expended or verify that the health conditions of the user/patient are within acceptable bounds.

The main technical contribution of the paper is to suitably modify and partition standard signal processing algorithms in order to retain the original recognition and monitoring quality and at the same time save power consumption. We show that we can achieve power savings of upto 20.6% for the full system (and 26.7% for the mobile phone alone) while retaining a comparable quality of recognition and monitoring.

The rest of the paper is organized as follows. In the next section we give an overview of the standard signal processing techniques used for activity recognition and how we partition them. Our results are presented in Section 3. Finally, we conclude in Section 4 by outlining some directions for future work.

2 Physical Activity Recognition and Monitoring

Physical activity recognition using accelerometer data fits into the larger framework of context awareness and can therefore be considered as a classification problem [6]. Research efforts in the area of activity recognition have previously focused on activity classification using feature vectors in conjunction with classifiers like Naive Bayes (NB), Decision Tree (DTr), Decision Table (DT), Gaussian



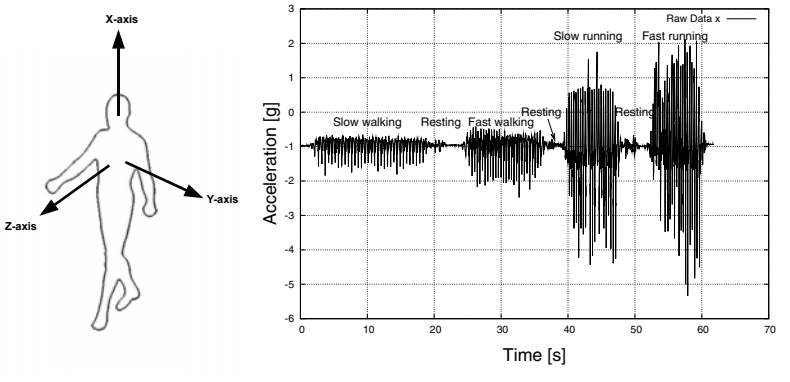
(a) Walking **(b)** Running

Fig. 1. Accelerometer data pattern for different activities

Mixture Model (GMM) and k-Nearest Neighbors (KNN) [5,6,3,2]. They have produced results with varying accuracies. In this paper we do not employ the above pattern recognition tools. Instead, our proposed algorithm makes use of pre-identified parameters in the acceleration signals in order to estimate the activity *type*. The basic concept is to design a flexible algorithm that uses a set of parameter constraints for the recognition task. The accuracy and robustness of the algorithm may be increased through additional constraints if such a need arises. This means that the accuracy of the algorithm can be adapted accordingly when the environment in which the algorithm operates, changes.

2.1 Nature of Acceleration Signals for Running and Walking

Figure 1 shows typical measurement values captured by a triaxial accelerometer during *Walking* and *Running* with the direction of the acceleration axis given in Figure 2a. In Figure 2b the x-acceleration data for several activity types is shown as an overview in order to give an impression of the obvious differences.



(a) Acceleration axis **(b)** X-acceleration data for several activity types

Fig. 2. Acceleration directions and activity types

Algorithm 1. Activity Classification (Mobile Phone)

```
   Input: x-acc[64], y-acc[64], z-acc[64]
   Output: state, steps[]
   /* Resting?                                                              */
 1 if variance (x-acc[64]) < x-threshold
 2 and variance (y-acc[64]) < y-threshold
 3 and variance (z-acc[64]) < z-threshold then
 4 |   print ("REST")
 5 |   steps[] ← 0
 6 |   state ← rest
 7 else
   |   /* Walking or Running?                                              */
 8 |   fft-coeff[] ← fourier-transform (x-acc[64])
 9 |   if max (fft-coeff[]) > threshold_{x-spec} then
   |   |   /* Perform step-detection with k-clustering algorithm           */
10 |   |   steps[] ← step-detection (x-acc[64])
11 |   |   if steps[] ≠ 0 then
12 |   |   |   print ("RUNNING")
13 |   |   |   state ← running
14 |   |   end
15 |   else
   |   |   /* Check, if periodic pattern in z-acceleration and if variance of
   |   |      y-acceleration exceeds threshold                             */
16 |   |   fft-coeff[] ← fourier-transform (z-acc[64])
17 |   |   if fft-coeff[] ≠ 0 and variance (y-acc[64]) > y-threshold-walking then
   |   |   |   /* Perform step-detection with k-clustering algorithm       */
18 |   |   |   steps[] ← step-detection (z-acc[64])
19 |   |   |   if steps[] ≠ 0 then
   |   |   |   |   /* General Walking, distinguish different walking types here... */
20 |   |   |   |   print ("WALKING")
21 |   |   |   |   state ← walking
22 |   |   |   else
   |   |   |   |   /* Do further checks here (movments, but no steps...)  */
23 |   |   |   end
24 |   |   else
   |   |   |   /* Do further checks here (movements, but not in y-axis...) */
25 |   |   end
26 |   end
27 end
28 return state, steps[]
```

The principle behind our algorithm is to define relevant parameters that allow us to effectively distinguish between different activity types. A total of eight such parameters have been defined. Algorithm 1 gives a schematic overview of how different activity types are classified. First, the variances of the data corresponding to the three axes are calculated, and if they are below a certain threshold then the activity is classified as *resting*. The next, more important parameter in determining the activity class, is the $threshold_{x-spec}$ parameter in the spectral domain. This is because $threshold_{x-spec}$ enables us to differentiate between walking and running. With parameters from the spectral domain [5], we are able to determine whether predefined frequency components above certain amplitudes exist in the frequency spectrum (as it may be seen in Figure 3). As the frequency spectrum is computed for the x-acceleration data, we do not filter this data because the x-acceleration data contains distinctive periodic patterns that are used for a peak detection algorithm to work effectively. This is why the

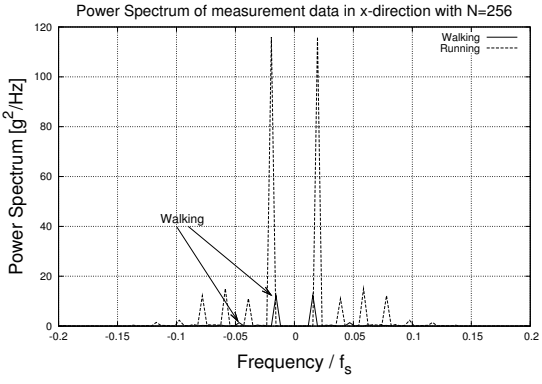


Fig. 3. Power spectrum of acceleration data in x-direction

x-acceleration data (and not the acceleration data from the remaining two axes) is crucial for detecting the activity type *running*.

We have observed that the value $threshold_{x-spec}$ has a relatively low variance across all test subjects we examined. Hence, $threshold_{x-spec}$ may be considered a relatively stable parameter. If frequency components above $threshold_{x-spec}$ are present in the spectrum, the data frame will subsequently be checked for running steps. This is possible because running motions result in a unique low frequency component in the x-acceleration spectrum that has a magnitude which is clearly above $threshold_{x-spec}$. With a k-clustering algorithm, the exact timestamp when a step occurs, is determined and the time between steps may also be calculated. If the frequency components are below $threshold_{x-spec}$, a different type of activity is ongoing, which needs to be analyzed further.

Similar to the first part of the analysis, the next part relies on the identification and definition of relevant threshold values in order to distinguish between different *walking* types. Here, spectral threshold values are used to identify frequency bins that are present above predefined threshold values. If the relevant frequency components are detected, then peaks and troughs detection is activated in order to detect the walking steps.

For walking, the important threshold values are $threshold_{z-spec}$ and th_{var-w}. The first threshold value $threshold_{z-spec}$ is similar to $threshold_{x-spec}$. It serves the same purpose as $threshold_{x-spec}$ with the difference that the spectrum being used for detecting walking belongs to the z-acceleration instead of the x-acceleration. The z-acceleration is used here because the x-acceleration spectrum for walking contains side lobes that are comparatively higher in magnitude than the z-acceleration spectrum. As a result, using the z-acceleration spectrum results in a better detection rate for walking steps. The second threshold value th_{var-w} is included in the algorithm to improve the detection results. It imposes the condition that the spread or variance of the y-acceleration must be greater than the threshold value in order for the data to be considered for steps detection. First the spectrum is checked for frequency components above the first

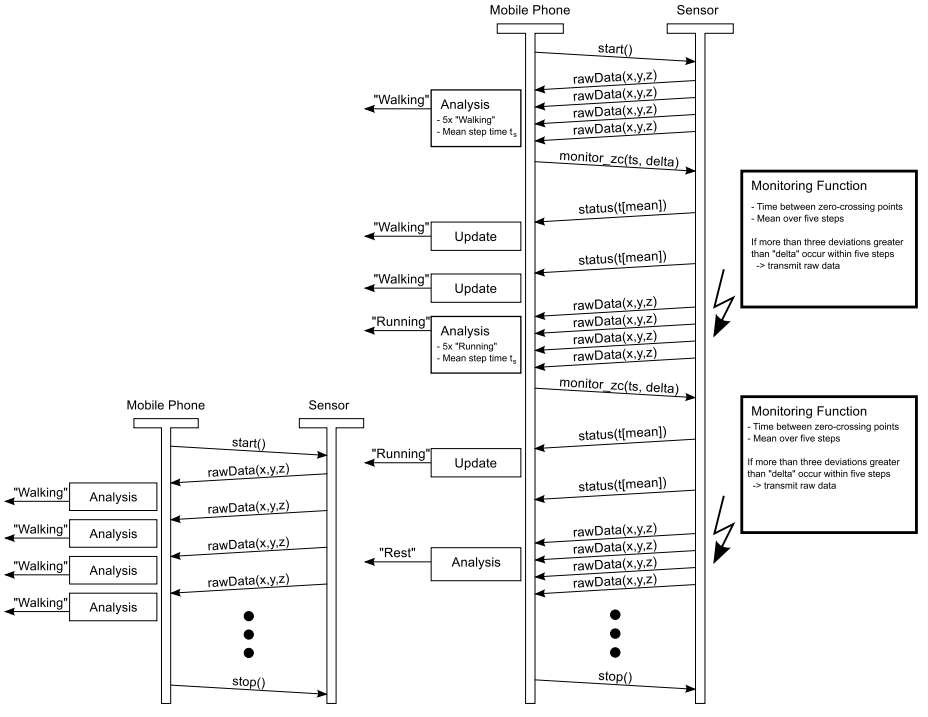


Fig. 4. Sequence charts for data transmission

threshold value. In addition to this, checks are performed so that the variance of the y-acceleration is greater than the second threshold value. If both conditions are met, the process for steps detection is activated. The threshold values for different population groups should be determined empirically as these parameters are quite stable but may depend on weight and height. The process of defining $threshold_{z-spec}$ can be seen as drawing a horizontal line across the spectrum and then checking for the existence of frequency bins above this line. This is a simple calibration technique that may be performed once before the user is equipped with this device for the first time.

2.2 Data Transmission

As already mentioned, our setup consists of a small sensor board measuring acceleration data at a sampling rate of 128 Hz, followed by transmitting it via Bluetooth. The analysis described in Section 2.1 is performed on the mobile phone each time new data packets arrive. This means that every 500 ms new data (64 values in each direction) is available and the activity classification has to be updated as shown in Figure 4a. This update rate has to be fulfilled in soft real-time for several reasons.

1. the sensor has a small amount of memory, and hence a limited amount of sensor data may be buffered,
2. the sensor has limited computational capability and energy budget and hence it is not able to handle all the computation involved,
3. often it is necessary to pass the results on to higher-level applications or to transmit them over the mobile network,
4. the user wants to get instantaneous feedback on the results of the monitoring.

The abovementioned procedure has the disadvantage of constantly engaging the mobile phone and thereby severely depleting its battery life. On the other hand, the sensor node is under-utilized, implying that it may also be used to share some of the computational tasks.

2.3 Cooperative, Distributed Analysis for Energy Optimization

The primary objective of the physical activity monitoring application is to automatically recognize different physical activities and record the relevant data for subsequent reprocessing. To achieve this a few parameters are necessary and have to be initially stored. These parameters are:

- type of physical activity
- intensity of execution
- duration

In [1] different types of physical activities are listed together with their corresponding MET[1] values. These MET values depend heavily on the specific physical activity type, e.g., walking, running, or cycling, and their performed intensity. If these MET values are known, it is then possible to determine the energy expenditure by multiplying them with their appropriate durations. An important observation is that during walking or running it is not necessary to precisely determine the duration of each step. Rather, it is sufficient to estimate the mean step duration as a base value and use it for calculating the walking or running speed and intensity. However, the main problem is the volume of data that needs to be transmitted between the sensor node and the mobile phone since the sensor node alone cannot support the computation necessary for estimating the necessary parameters.

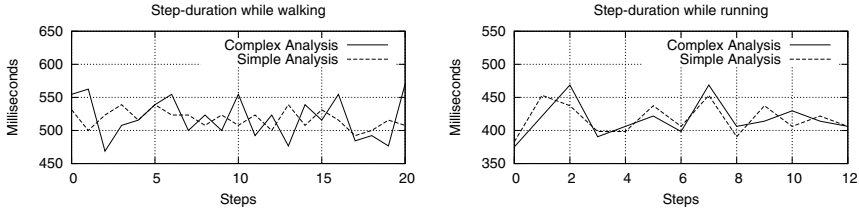
Our main observation is that the recorded sensor data is periodic in nature. If one is walking, the *shape* of the acceleration data recorded from one step is similar to the next one, provided the walking speed does not change significantly. The duration of each step can easily be computed by searching the x-acceleration data for the *zero-crossing* points with an offset of -1 g. This is shown in Algorithm 2. This algorithm can be used both for *walking* and *running* data. However, it may not be suitable to *distinguish* between these activities in a reliable fashion and nor can it *recognize* the activity itself. The solution to this problem is to perform the *recognition* task on the mobile phone. Once a periodic pattern like walking or running with a nearly constant speed is recognized, this information is

[1] Metabolic Equivalent.

Algorithm 2. State tracking (Sensor)

```
   Input: x-acc_old, x-acc_new, threshold, time_SetPoint, time_delta
   Output: steps[]
 1 if mode == zcPoint then
       /* Search crossing-points                                    */
 2     if x-acc_old < threshold and x-acc_act > threshold then
           /* Determine time between last and new crossing-point    */
 3         time_new = GetTime()
 4         time = time_act − time_old
 5         time_old = time_new
           /* Check deviation                                       */
 6         if abs (time- time_SetPoint) > time_delta then
 7             mode ← rawData
 8         else
 9             steps[] ← time
10             if Number of steps == 5 then
11                 SendStatus (steps[])
12                 clear steps[]
13             end
14         end
15     end
16 else
17     SendRawData()
18 end
```

transmitted to the sensor node and from there on the sensor node can take over the *monitoring* function. As soon as the sensor node takes over the monitoring task (along with the set-point and the maximum allowed difference from this set-point), it is able to act completely on its own. Only a *status update* is sent to the mobile phone once every five steps, containing either the step durations themselves or their mean value. Figure 4b shows the corresponding sequence chart. In summary, after a *steady-state* in the activity *recognition* is reached, the sensor node gets all the necessary information to do the *monitoring*. If the difference between the measured value and the set-point is greater than a fixed variance, raw data is transmitted and the entire process starts all over once again. With such a cooperative and distributed analysis, only the computationally expensive activity recognition task is performed on the mobile phone and the less energy consuming monitoring task is performed on the sensor node, thereby avoiding the need to transmit all the measured data. Note that the time between two zero-crossing points with an offset of -1g is a necessary but not sufficient condition for activity recognition. However, in conjunction with the apriori knowledge generated by the mobile phone, it is sufficient to track a physical activity. Without this knowledge the sensor node can no longer rely on a single parameter and has to carry out a more detailed computation involving several parameters (thereby requiring more energy and computational capacity).



(a) Step duration (*Walking*) **(b)** Step duration (*Running*)

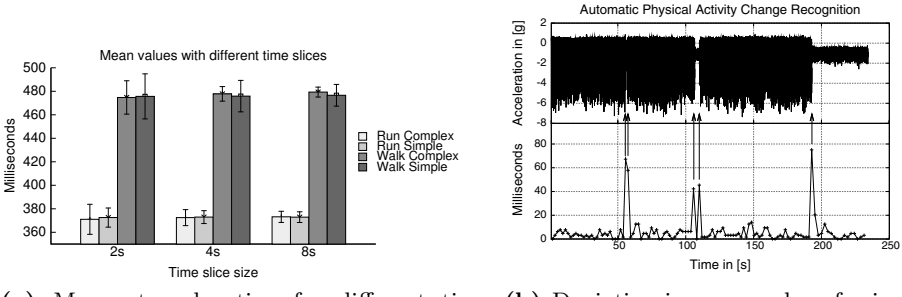
Fig. 5. Step time comparison

3 Experimental Results

In this section we compare the physical activity recognition and monitoring results obtained from the (i) purely mobile phone based implementation, and (ii) from the distributed, cooperative implementation. We also compare the energy consumptions of these two implementations. In the following discussion, the raw-data transmission and subsequent analysis on the mobile phone is referred to as the *complex* analysis/algorithm because of the multiple parameters that are used by the algorithm (as shown in Algorithm 1). On the other hand, the distributed zero-crossing Algorithm 2 is referred to as *simple* because of the comparatively fewer parameters its uses.

Figures 5a and 5b show the times between successive steps in milliseconds. Ideally, the measured step durations should be the same, but in reality small differences can be seen. During *Walking* the time between successive steps is not as similar as those during *Running*. This depends on the fact that acceleration patterns during running are more periodic and self similar compared to those during walking.

As mentioned before, not single step durations, but rather the *mean* value of these durations is important for physical activity recognition. Such mean values may either be computed over a number of steps (e.g. 5 steps) or over fixed time



(a) Mean step duration for different time slices **(b)** Deviation in mean values for interleaved *Walking* and *Running*

Fig. 6. Mean step durations for *Walking* and *Running*

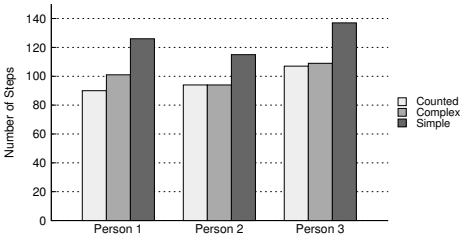


Fig. 7. Number of steps counted by visual inspection versus estimated using the proposed complex and simple algorithms

durations or slices (e.g. 3 sec). Figure 6a shows such mean values and illustrates a good conformity between the results obtained from the complex and the simple algorithms. It may be seen that the differences between the results from the complex and the simple algorithms decrease for longer time slices. In order to obtain Figure 6a, the mean value of step durations within intervals of two, four and eight seconds have been used. Here, all step durations within two, four or eight seconds have been recorded and their mean values have been computed. After this, the next time slice is used for computing the next mean value. In this manner a list of mean values is obtained. Finally, the mean of all these mean values is computed, together with the corresponding variance (as shown in Figure 6a). The entire data set consisted of four minutes of *Walking* and an additional four minutes of *Running*.

As soon as a change in physical activity occurs, the sensor switches its transmission into the raw mode. This change may be clearly seen in Figure 6b. In the upper half of the figure, the raw data from the x-acceleration axis has been shown (as in Figure 2b) and the interleaving between *Walking* and *Running* may be seen. The lower half of the figure shows the corresponding difference between two adjacent mean values. If this value is too high then it implies a change in physical activity (e.g., from *Walking* to *Running*).

Next we show the correctness of the number of estimated steps. As it can be seen in Figure 7, the complex physical activity recognition algorithm almost correctly estimates the number of steps, when compared with counting by visual inspection. Here, the data set was obtained from three different persons walking and running the same distance. All values from the same person were analyzed in order to see if the simple algorithm is able to distinguish between *Walking* and *Running* correctly. The simple algorithm, without any apriori knowledge, detects the number of steps using the deviation from mean step duration technique (with the person occassionally in the *Resting* mode). Here, the acceleration values oscillate around the offset/threshold value, resulting in the detection of "phantom" steps. This is the reason for the increased number of detected steps. The conclusion from Figure 7 is that the complex algorithm correctly detects phases without movements and during such phases does not detect steps, whereas the simple algorithm always tries to find one. However, when there is movement, the number of detected steps are within a comparable range.

The major benefit of the simple algorithm is the reduced energy consumption, especially in the mobile phone. If the simple algorithm completely executed on the mobile phone, then every measurement value would have to be transmitted and the mobile phone would not be able to get into the sleep mode.

$$E_{total}(t) = E_{data\_acq}(t) + E_{mcu}(t) + 2 \times E_{bt}(t) + E_{phone}(t) \quad (1)$$

Equation (1) shows the different components comprising the total energy consumed by the setup. First, is the energy consumption as a result of data acquisition (E_{data_acq}). The second term corresponds to the preprocessing on the microcontroller of the sensor node (E_{mcu}), the third term arises of the bluetooth communication (E_{bt}) and finally the last term arises from the analysis on the mobile phone (E_{phone}). The energy consumption of the data acquisition part (E_{data_acq}) depends on the circuit design and therefore cannot be optimized during runtime (it is static).

The energy consumed during data transmission (E_{bt}) is static too, but it may be reduced by transmitting less data. Such a reduction is achieved through the proposed cooperative distributed processing. We next give some numbers corresponding to the current consumed by the different subparts of our setup. The data acquisition part consumes about 0.65mA, the microcontroller about 2mA and the bluetooth module during transmission approx. 30mA and 2.2mA during connection in the *sniff* mode (both on the sensor node and on the mobile phone). All of these components work with the same supply voltage of 3.3V. Therefore, the energy consumption depends on the current consumption and the associated running time. The mobile phone consumes about 260mW in active mode, and only 30mW during idle mode, when the processor is sleep state. The proposed approach for energy saving relies on minimizing the amount of data to be transmitted and analyzed.

Energy consumption during data transmission is directly related to the amount of data and hence the necessary transmission time t_{bt_txrx}. This time is needed to transmit the necessary data between the sensor node and the mobile phone, where the running time for the analysis is crucial too. In the following, power consumption associated with the two cases – complex and simple – are analyzed.

Complex Analysis: In the complex case (i.e., without data analysis on the sensor node) all measurement values have to be transmitted via bluetooth to the mobile phone. Using a sample rate of $f_s = 128$Hz and two bytes per acceleration axis we get 768 bytes/sec. This has to be split into 400 byte packets, including additional informations like timestamp and packet number (16 bytes), because the sensor node does not have enough memory to buffer measurement values for one second. These packets are transmitted every 500ms. With a max. transfer rate of 721kbit s^{-1} the transmission lasts for

$$t_{bt\_txrx} = \frac{8 \times 800 bytes}{721 kbit\, s^{-1}} = 8.9ms$$

per two packets, during every second. Each time a new packet arrives on the mobile phone, the analysis algorithm is invoked. This requires approx. 16ms per run. With this information, the energy consumption over a one second period may be calculated as follows:

$$\begin{aligned}
&E_{complex}(1s) = 3.3V \times 0.65mA \times 1000ms(E_{data_acq}) + 3.3V \times 2mA \times 1000ms(E_{mcu}) \\
&+ 2 \times 3.3V \times (30mA \times 8.9ms + 2.2mA \times 991.1ms)(E_{bt}) \\
&+ 260mW \times (8.9ms + 2 \times 16ms)(E_{phone}, active) + 959.1ms \times 30mW(E_{phone}, idle) \\
&= \mathbf{64.3mWs}
\end{aligned}$$

Simple Analysis: With the optimized analysis on the sensor node, only a fraction of the data has to be transmitted to the mobile phone since the analysis is done on the sensor. With *Walking* or *Running* the time for five steps is about 1.5–2 seconds. With this, we have the same packet overhead of 16 bytes per packet and we have to transmit 2 bytes per step. For every five steps a packet consisting of $(5\times\ 2 + 16)$ bytes = 26 bytes is transmitted. This occurs every 1.5 seconds, so we get a transmission time of:

$$t_{bt\_txrx} = \frac{8 \cdot 26/1.5\text{bytes}}{721kbit\,s^{-1}} = 0.19ms$$

over every second. From this, the energy consumption may be calculated over a one second period in an analogous way:

$$\begin{aligned}
&E_{simple}(1s) = 3.3V \times 0.65mA \times 1000ms(E_{data_acq}) + 3.3V \times 2mA \times 1000ms(E_{mcu}) \\
&+ 2 \times 3.3V \times (30mA \times 0.19ms + 2.2mA \times 999.81ms)(E_{bt}) \\
&+ 260mW \times (0.19ms + 2 \times 0.05ms)(E_{phone}, active) + 999.71ms \times 30mW(E_{phone}, idle) \\
&= \mathbf{53.3mWs}
\end{aligned}$$

These numbers show that the complex analysis needs $\frac{64.3-53.3mWs}{53.3mWs} = 20.6\%$ more energy compared to the simple algorithm. As already mentioned, the bottleneck in the overall system is the energy consumption in the mobile phone, which is composed of the consumption at the processor and in the Bluetooth module. In the mobile phone, the complex algorithm needs $\frac{47.5-37.5mWs}{37.5mWs} = 26.7\%$ more energy compared to the proposed simple one.

4 Concluding Remarks

In this paper we have proposed distributed, cooperative signal processing algorithms for physical activity recognition and monitoring. Our proposed algorithms perform a considerable amount of processing on the sensor node itself, thereby reducing the volume of data to be communicated to the mobile phone. Processing all the data on the mobile phone – using standard signal processing algorithms – require nearly 26% higher energy consumption. As a part of future work, we will investigate individual-specific dynamically adapted thresholds (in contrast to static ones) for better recognition quality.

References

1. Ainsworth, B.: The compendium of physical activities tracking guide (January 2002), http://prevention.sph.sc.edu/tools/compendium.htm
2. Bao, L., Intille, S.S.: Activity recognition from user-annotated acceleration data. In: Ferscha, A., Mattern, F. (eds.) PERVASIVE 2004. LNCS, vol. 3001, pp. 1–17. Springer, Heidelberg (2004)
3. Bidargaddi, N., Sarela, A., Klingbeil, L., Karunnanithi, M.: Detecting walking activity in cardiac rehabilitation by using accelerometer. In: 3rd International Conference on Intelligent Sensors, Sensor Networks and Information (December 2007)
4. Diemer, R., Kreuzer, J.: Multi-Sensorplattform for Activity Measurements. In: International Conference on Ambulatory Monitoring of Physical Activity and Movements (2008)
5. Ibrahim, R.K., Ambikairajah, E., Cellar, B.G., Lovell, N.H.: Time-frequency based features for classification of walking patterns. In: 15th International Conference on Digital Signal Processing (July 2007)
6. Ravi, N., Dandekar, N., Mysore, P., Littman, M.L.: Activity recognition from accelerometer data. American Association for Artificial Intelligence (2005)

Author Index