American Journal of Computer Architecture 2012, 1(2): 21-36
DOI: 10.5923/j.ajca.20120102.02

Architectural Optimization & Design of Embedded Systems based on AADL Performance Analysis

Roberto Varona-Gómez[1,*], Eugenio Villar[1], Ana Isabel Rodríguez[2], Francisco Ferrero[2], Elena Alaña[2]

[1]University of Cantabria, 39005, Santander, Spain
[2]GMV Aerospace and Defence S.A.U., 28760, Tres Cantos, Spain

Abstract Due to the increasing complexity of embedded systems, new design methodologies have to be adopted, since traditional techniques are no longer efficient. Model-based engineering enables the designer to confront these concerns using the architecture description of the system as the main axis during the design cycle. Defining the architecture of the system before its implementation enables the analysis of constraints imposed on the system from the beginning of the design cycle until the final implementation. AADL has been proposed for designing and analyzing SW and HW architectures for real-time mission-critical embedded systems. Although the Behavioural Annex improves its simulation semantics, AADL is a language for analyzing architectures and not for simulating them. AADS is an AADL simulation tool that supports the performance analysis of the AADL specification throughout the refinement process from the initial system architecture until the complete, detailed application and execution platform are developed. In this way, AADS enables the verification of the initial timing constraints during the complete design process. AADS supports the performance analysis of the AADL specification, enriched with behaviour specifications. AADS-T is Ravenscar Computational Model (RCM) compliant as part of the TASTE toolset and has been used to assist in co-design.

Keywords AADL, Performance Analysis, Simulation, Ravenscar Computational Mode, HW/SW Co-Design, AADS, POSIX, SCoPE

1. Introduction

Nowadays, embedded systems must support the deployment of heterogeneous applications within heterogeneous architectures. In most cases, the execution platform is not fixed and must be designed and optimized in conjunction with the application SW. Therefore, early estimation of the system performance on the executive platform, under real-time constraints, is desirable. This analysis requires a unified model of the application and the architecture, and effective means to define the mapping of application functions onto architecture resources and services.

Architecture Analysis and Design Language (AADL) [1-3] provides such a modelling framework. It was developed as a standard of the Society of Automotive Engineers (SAE) to enable the description of task and communication architectures for real-time, embedded, fault-tolerant, secure, safety-critical, SW-intensive systems. However, AADL does not support the expression of behaviour in detail. At most, it is possible to specify the non-deterministic behaviour of a thread as a set of subprogram calls, and application behaviour relies mainly on source code written in source languages. The behavioural annex[4] has introduced high-level composition concepts and a richer state representation than the standard AADL mode automata[48]. The behaviour is specified using extended automata that may trigger a transition by an event, a Boolean expression, etc. A transition may trigger one or more actions such as assignment of values to variables, sending data, events, etc. The annex mainly declares states and transitions with guards and an action part. Guards and actions can access ports and data subcomponents declared in the AADL component to which they are attached.

The Automated proof-based System and Software Engineering for Real-Time systems (ASSERT) project[5] resulted in a new development process for distributed embedded real-time software, and a set of methods and tools supporting the process. The process is based on separation of concerns, automatic code generation and property preservation. An important feature of the ASSERT process is the adherence of the concurrency model to the RCM[6], a restricted tasking model that enables static response time analysis of real-time systems. The model restricts the concurrency model to a static set of periodic and sporadic threads communicated by means of a static set of shared data objects, protected by mutual exclusion

* Corresponding author:
roberto@teisa.unican.es (Roberto Varona-Gómez)
Published online at http://journal.sapub.org/ajca

synchronization. There are two variants of the ASSERT software process: Hard Real-Time Unified Modeling Language (HRT-UML) and AADL tracks. The ASSERT Set of Tools for Engineering (TASTE)[7] toolset is an open source toolset supporting the latter.

The LEON2[8] processor was designed by the European Space Agency (ESA) as a 32-bit synthesizable processor core based on the SPARC V8 architecture. The core is highly configurable, and particularly suitable for System-on-Chip (SOC) designs.

There is a commonly recognized need for new development frameworks that enable designers to perform efficient exploration of design alternatives and analyze system properties throughout the design cycle. Some system properties can be obtained by static analysis. Many other properties can only be obtained through simulation. In any case, system simulation is needed for performance analysis under real execution conditions. System simulation and performance analysis can validate the correct dimensioning of the system and detect locks, missed deadlines and other potential problems raised by the complex interaction among components that can be found in a real system. The earlier all these problems are detected, the lower the cost of correcting them[9].

Evolutionary prototyping is now becoming a well-accepted development approach in Model-Driven Engineering (MDE)[10]. The design flow is based on a central model that is refined unless it is satisfactory. Programs can be generated from this model and constitute intermediate versions of the product. The last refined model corresponds to the final system. A prototyping-based design process is beneficial for the earliest possible verification of the impact of deployment decisions, or the use of a particular HW/SW component in the system.

This document deals with AADS[12] an AADL simulation and performance analysis framework, including a behavioural annex compatible with the RCM. The tool can support prototype-based design allowing the functional and non-functional (execution times, power consumption, etc.) verification of the system while it is being refined until the final implementation. Based on SystemC, the framework supports the seamless integration of HW component and an easy optimization of the executive platform. SystemC has become a relevant standard language for modeling and simulation of HW/SW embedded systems[11].

HW/SW partitioning is a phase of co-design in which the partition of the specification into two parts is achieved. One part will be implemented in HW and the other part will be implemented in SW. HW/SW partitioning is the process of deciding, for each subsystem, whether the required functionality is more advantageously implemented in HW or SW to achieve a partition that will give us the required performance within the overall system requirements (in size, weight, power, cost, etc.). Partitioning into HW and SW affects overall system cost and performance[45]. There are two partitioning approaches: Starting first with all the functionality in SW and moving parts, which are time-critical and cannot be allocated to SW, into HW (this is known as SW centric partitioning) or starting first with all the functionality in HW and moving parts into the SW implementation (this is known as HW centric partitioning). AADS extracts the necessary information for the SCoPE tool to perform a simulation at system level from the AADL models. The simulation and performance analysis results will guide the system designer through the selection of the most adequate partition solution (see Figure 1).

The contents of the paper are as follows. The following two sections make a summary of AADL and SCoPE respectively. The next section reviews the related work. In the next two sections, AADS and AADS-T are described, with a brief description about performing the simulation on a LEON2 processor. Next, we explain the Case Study: How AADS-T has been used in the ESTEC 22810/09/NL/JK HW-SW CODESIGN project[46] to assist in HW/SW partitioning. Then we state the conclusions and finally, we include acknowledgements and referenced documents.

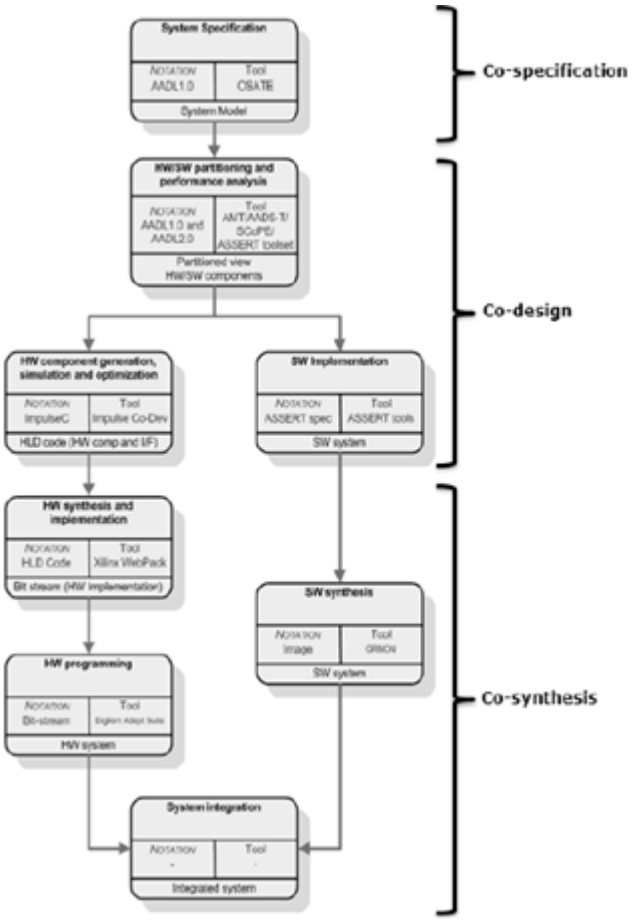

Figure 1. AADS and SCoPE in the HW/SW co-design process

2. AADL

The SAE AADL standard provides formal modelling concepts for the description and analysis of application system architecture in terms of the distinct components and their interactions. The AADL includes software, hardware,

and system component abstractions to specify and analyze real-time embedded systems, complex systems of systems, and specialized performance capability systems, and to map software onto computational hardware elements. The AADL is especially effective for model-based analysis and specification of complex real-time embedded systems.

In AADL, a component is characterized by its identity (a unique name and runtime essence), possible interfaces with other components, distinguishing properties (critical characteristics of a component within its architectural context), and subcomponents and their interactions. In addition to interfaces and internal structural elements, other abstractions can be defined for a component and system architecture. For example, abstract flows of information or control can be identified, associated with specific components and interconnections, and analyzed. These additional elements can be included through core AADL language capabilities (e.g. defining new component properties) or the specification of a supplemental annex language.

The component abstractions of the AADL are separated into three categories: Application software, execution platform (hardware) and composite. Application software can be a thread (active component that can execute concurrently and be organized into thread groups), thread group (component abstraction for logically organizing thread, data, and thread group components within a process), process (protected address space whose boundaries are enforced at runtime), data (data types and static data in source text) and subprogram (concepts such as call-return and calls-on methods, modelled using a subprogram component that represents a callable piece of source code). Execution platform (hardware) can be a processor (schedules and executes threads), memory (stores code and data), device (represents sensors, actuators, or other components that interface with the external environment) and bus (interconnects processors, memory, and devices). Composite can be a system (design elements that enable the integration of other components into distinct units within the architecture). System components are composites that can consist of other systems as well as of software or hardware components.

The AADL standard includes runtime semantics for mechanisms of exchange and control of data, including message passing, event passing, synchronized access to shared components, thread scheduling protocols, timing requirements and remote procedure calls. In addition, dynamic reconfiguration of runtime architectures can be specified using operational modes and mode transitions.

The AADL can be used to model and analyze systems already in use and design and integrate new systems. The AADL can be used in the analysis of partially defined architectural patterns (with limited architectural detail) as well as in full-scale analysis of a complete system model extracted from the source code (with completely quantified system property values).

AADL supports the early prediction and analysis of critical system qualities, such as performance, schedulability, and reliability. For example, in specifying and analyzing schedulability, AADL-supported thread components include the pre-declared execution property options of periodic, aperiodic (event-driven), background (dispatched once and executed until completion), and sporadic (paced by an upper rate bound) events. These thread characteristics are defined as part of the thread declaration and can be readily analyzed.

Within the core language, property sets can be declared that include new properties for components and other modelling elements (e.g. ports and connections). By utilizing the extension capabilities of the language, additional models and properties can also be included. For example, a reliability annex can be used that defines reliability models and properties of components facilitating a Markov or fault tree analysis of the architecture. This analysis would assess architecture's compliance with specific reliability requirements.

Collectively, these AADL properties and extensions can be used to incorporate new and focused analyses at the architectural design level. These analyses facilitate trade off assessments among alternative design options early in a development or upgrade process.

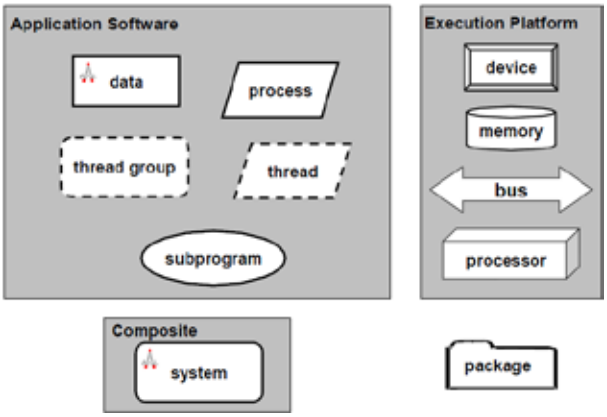

Figure 2. AADL graphical notation

AADL components interact exclusively through defined interfaces. A component interface consists of directional flow through data ports for un-queued state data, event data ports for queued message data, event ports for asynchronous events, synchronous subprogram calls and explicit access to data components.

Interactions among components are specified explicitly. For example, data communication among components is specified through connection declarations. These can be mid-frame (immediate) communication or phase-delayed (delayed) communication. The semantics of these connections assures deterministic transfer of data streams. Deterministic transfer means that a thread always receives data with the same time delay; if the receiving thread is over- or under-sampling the data stream, it always does so at a constant rate.

Application components have properties that specify timing requirements such as period, worst-case execution

time, deadlines, space requirements, arrival rates, and characteristics of data and event streams. In addition, properties identify source (code and data that implement the application component being modelled in the AADL) and constraints (for binding threads to processors, source code, and data onto memory). The constraints can limit binding to specific processor or memory types (e.g., to a processor with DSP support) as well as prevent co-location of application components to support fault tolerance.

3. SCoPE

The SCoPE[24] tool provides the technology to perform MPSoC HW/SW co-simulation with Network on Chip (NoC). It enables the exploration of the design space to choose the right processors and HW/SW partition for embedded systems. It also allows the simulation of different nodes connected through a NoC in order to analyse the behaviour of large systems. Commonly, these tools are based on slow ISSs. The differentiating feature of this technique is that SCoPE obtains the performance estimations at source code level. This level of abstraction enables the simulation time to be reduced significantly while maintaining good accuracy.

SCoPE is a C++ library that without modification extends standard language SystemC to perform co-simulation. On the one hand, it simulates C/C++ software code based on two different operating system interfaces (POSIX and MicroC/OS). On the other hand, it co-simulates these pieces of code with hardware described in SystemC.

An engineer with this tool can simulate specific software over a custom platform and obtain estimations of: number of thread and context switches, running time and use of CPU, instructions executed and cache misses, energy and power (of core and instruction cache).

This library models the detailed behaviour of the RTOS including concurrency (among tasks in the same processor), parallelism (among tasks in different processors), scheduling and synchronization. Although the SystemC kernel executes processes following a non pre-emptive scheduling policy without priorities, SCoPE models pre-emption under different scheduling policies based on priorities.

SCoPE integrates a POSIX-based API that enables the execution of a large number of software applications that follows this standard. POSIX is the main operating system interface nowadays, but it is not the only one. Thus, SCoPE has been improved to support extensions for other types of interfaces. An example is the integration with the MicroC/OS interface. This is a demonstration of the scalability of the tool, in terms of software support.

The design of embedded systems requires not only software handling but also hardware communication. For this reason SCoPE includes a set of more than a hundred driver facilities to implement this communication. One of the most extensively used operating systems in this sector is Linux, so these driver facilities are based on the Linux kernel version 2.6. Furthermore, SCoPE is able to simulate the loading of kernel modules and the handling of hardware interruptions and their corresponding scheduling.

SystemC is the language used for the modelling of the hardware platform due to the easiness of implementation (C++ extension) and its simulation kernel. For the purpose of simulating different platforms SCoPE incorporates some generic hardware modules: A bus based on TLM2 used for the communication with peripherals and the transmission of hardware interruptions, a DMA for copying large amounts of data, simple memory for the simulation of cache and DMA traffic, a hardware interface for simple custom hardware connection, a network interface that works as a net card for the NoC and an external network simulator to implement the NoC connected to SCoPE.

System simulation comprises Multi-computation and Modular structure. Multi-computation: One of the advantages of this tool is the possibility of interconnection among independent nodes and simulating the interaction among them. Modular structure: Each RTOS component is an independent object that does not share any data with the others. Furthermore, each process is isolated from the rest of the system, thus a process with global variables can be replicated in many nodes without data collision problems. That is, each process has a separate memory space.

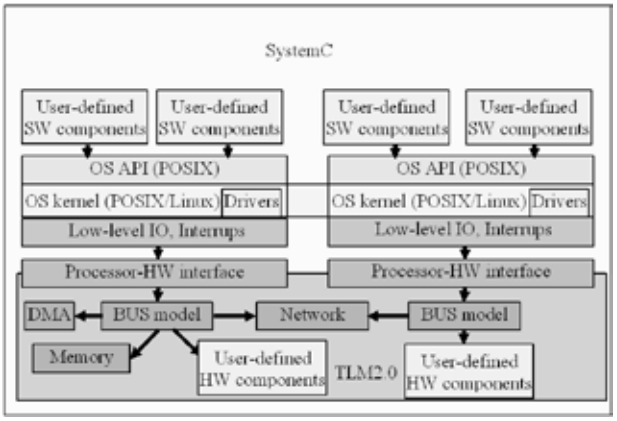

Figure 3. Block diagram of SCoPE

4. Related Work

Simulation and performance analysis of AADL models represent an important stage in MDE. Different approaches address this issue.

ADeS is one of the most powerful simulation tools yet it requires the environment in which the system evolves[15] to be taken into account.

Another way to tackle the problem is translating AADL to another language. Cheddar[16] is a set of Ada packages that enables the design of a new scheduler and direct interpretation using the Cheddar environment. The Furness toolset[17] translates models into the real-time process algebra ACSR to explore the state space looking for

violations of timing requirements. M. Yassin Chkouri et al. propose in[18] a translation from AADL models to BIP models to enable simulation. Ocarina[10] is a tool suite that uses code generation facilities in Ada and C to analyze the AADL model. ADAPT[19] translates an AADL architectural model into a dependability evaluation model in the form of a Generalized Stochastic Petri Net (GSPN). T. Abdoul et al.[20] produce an IF timed automata model which is the entry point of the validation process, processing it with the IFx framework. E. Jahier et al.[21] translate the architecture into a non-deterministic synchronous model to which the SW components in Scade or Lustre can be integrated, to simulate it with Lurette. Annex D of the AADL standard gives guidelines to translate AADL SW components into source code (C, Ada).

S. Gui et al.[22] use the linear hybrid automata in the design phase statically to abstract the semantics of the SW components of AADL explicitly.

M. Brun et al.[23] translate to OIL configuration code and to C code which is compatible with the OSEK/VDX RTOS.

Several authors have considered the behavioural annex in their research on AADL. Some of their papers were written in the initial stage of the behavioural annex so they were intended to evaluate, promote and disseminate it. P. Dissaux et al.[29] present a proposal for a behavioural annex to the AADL standard. They explain how to implement the behavioural annex with the Stood tool, a graphical AADL editor that can import and export AADL textual specifications. R. Bedin et al.[30] evaluate the behavioural annex through a flight software design in the ArchiDyn project. This requires new synchronization primitives for AADL runtime and support using edition and analysis tools for the behavioural annex. J. P. Bodeveix et al.[31] propose an AADL behavioural annex and a technique to perform compositional real-time verification of AADL models through the use of a method which translates environmental constraints into behaviour.

Other papers, such as the latter one, include the behavioural annex in their verification process of AADL models. B. Berthomieu et al. describe in[32] a formal verification tool chain for AADL with its behavioural annex available in the Topcased environment. They translate the AADL model to Fiacre and verify the behaviour with a Time Petri Net Analyzer (Tina).

C. Ponsard et al. explore in[33] the interplay of requirements and architecture in a model-based perspective by defining a mapping and a constructive process taking into account specifics of embedded systems, especially the importance of non functional requirements. To generate the behavioural part of a system they first generate a finite state machine and then an AADL mode-transition.

A way to approach AADL and its behavioural annex is translation to another language. To allow simulation M. Yassin Chkouri et al. propose in[34] a translation from AADL models to BIP models. They take into account behaviour specifications allowing state variables, initialization, states and transitions sections to be defined and translating them into BIP. DUALLY[35] is an automated framework that allows architectural language interoperability through automated model transformation techniques. I. Malavolta et al. analyze the feasibility of integrating AADL and OSATE in DUALLY. They map AADL behavioural annex sections of states, composite states and transitions.

Several authors have considered ASSERT and the RCM in their research. Some of their papers deal with the ASSERT Virtual Machine (VM), the execution platform on which ASSERT applications run, based on the RCM. J. A. de la Puente et al.[36] and J. Zamorano et al.[37] are good examples of this.

M. Bordin et al.[38] propose some guidelines to generate RCM-compliant Ada code from HRT-UML. S. Mazzini et al.[39] explain a MDE methodology for the development of high-integrity real-time systems. However using UML does not enable a low-level description of the system. Moreover, the different views of the system use different formalisms, so one must modify all views on each change of the system to get a coherent model, hindering rapid prototyping.

J. Kwon et al.[40] propose Ravenscar-Java, a high-integrity profile for real-time Java. However we think that Java is not a good high-integrity programming language due to its object-oriented programming features, its automatic garbage collection, and the proposed limitations to the extension of real-time multi-threading that cause confusion.

Ocarina[10] is a tool suite that uses code generation facilities in Ada and C to analyze AADL models. The code generated is compatible with the RCM.

In[50] M.B. Abdelhalim, S.E.-D. Habib develop a new high-level hardware/software partitioning methodology. In[51] K. Lampka et al. present a compositional and hybrid approach for the performance analysis of distributed real-time systems.

After analyzing the related work, it appears that no approach uses SystemC, which is a recognized standard for modelling HW/SW platforms, with its great potential for integration of processors, buses, memories and specific platform HW. The aforementioned solutions cannot model the HW platform so they do not permit HW/SW co-design. Apart from[23] none of the approaches models AADL over a RTOS. Our solution makes HW/SW co-design easier because of the use of SystemC.

SCoPE is a C++ library that extends standard language SystemC[25] without modifying it. It simulates C/C++ SW code based on two different operating system interfaces (POSIX[26-27] and MicroC/OS). Moreover, it co-simulates these pieces of code with HW described in SystemC. SCoPE generates a file with this SystemC description of the model. SCoPE has proven to be an efficient tool for design space exploration[49].

AADS supports AADL simulation in SystemC, thus allowing the modelling of the HW platform and permitting HW/SW co-design. The AADL model is based on POSIX,

so it supports many different RTOS.

In previous work[42-44] we have talked about AADS, nevertheless, in this paper we show the complete methodology with a complex industrial case study.

5. AADS

AADS is an AADL simulation tool written in Java, which was developed as a plug-in[13] of Eclipse[14]. AADS enables the modelling of a subset of AADL for purposes of implementation and simulation. The starting point of the simulator is a functional AADL specification without detailed code. For each component, the corresponding timing constraints are defined. This initial AADL specification supports the verification of the global performance constraints of the system based on the specific timing constraints of the different components. The AADL model is parsed using AADS and a model suitable for simulation with SCoPE is produced, in order to check whether the AADL constraints are fulfilled. As the design process advances and, on the one hand, the actual functionality is attached to the SW components using the corresponding source code and, on the other, the functionality is mapped onto specific platform resources, a more accurate performance estimation is achieved. These refined properties will be added to the AADL model and a new model is generated by AADS. By comparing the initial timing constraints with these refined, timing estimations, it is possible to verify the non functional correctness of the design process at any refinement step.

AADL enables the specification of both the architecture and functionality of an embedded real-time system. AADS translates both to SystemC (see Figure 4). It parses the AADL model so the functionality is translated to an equivalent POSIX model and the architecture is represented in XML[28]. The equivalent POSIX model and the architecture can be implemented easily as an embedded system.

The functionality of an embedded real-time system is translated as follows:

Threads. An AADL thread is a concurrent schedulable unit of sequential execution through source code and multiple threads represent concurrent execution paths. A POSIX thread is an execution thread in a program and an application can have multiple execution threads running concurrently. An AADL thread translates seamlessly into a POSIX thread.

In POSIX, a thread attribute object must be defined and initialized with the default value for all of the individual attributes used by a given implementation. AADS determines how the other scheduling attributes of the created thread are to be set, that is that the scheduling policy and associated attributes are to be set to the corresponding values. Thus AADS can now call the POSIX function to create a new thread with the specified attributes. The specified routine is then launched as a starting routine.

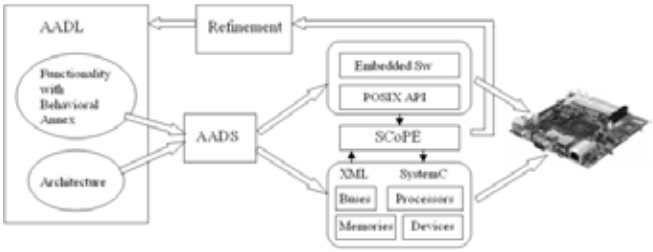

Figure 4. Translation, refinement and implementation with AADS

Periodic threads. A thread is periodic if repeated dispatches occur during a specific time interval. An AADL periodic thread has its Dispatch_Protocol property set to Periodic and its Period property set, for example, to 20 ms.

These two properties are translated putting the source code of the POSIX thread into an infinite loop. At the beginning of the loop the current time is obtained. At the end of the loop the current thread is suspended until either the time value of the clock reaches the absolute time specified (the current time plus the period), or a signal is delivered to the calling thread and its action is to invoke a signal-catching function, or the thread is terminated. By doing this it waits to repeat the loop for exactly the time specified in the Period property.

Port connections translate into message queues, signals and global variables:

Message queues. An AADL event data port models message communication with queuing of messages at the recipient. Message arrival may cause dispatch of the recipient and allow the recipient to process one or more messages. POSIX message queues allow threads to exchange data in the form of messages. Messages placed in the queue are stored until the recipient retrieves them. An AADL event data port connection between threads translates into a POSIX message queue between threads.

The attributes of the message queue must be set. The value of the maximum number of messages is taken from the AADL property Queue_Size of the destination port if it exists. The AADL property Queue_Processing_Protocol is set to FIFO as corresponds to a message queue. The message queue is created to both send and receive messages in non-blocking mode. The thread corresponding to the AADL source/destination thread of the event data port connection should add/receive a message of the specified length to/from the message queue specified with the priority indicated.

Signals. An AADL event port acts as an interface for the communication of events raised by subprograms, threads, etc. that may be queued. An example of use of an event port includes alarm communications that may be queued at the recipient, where the recipient may process the queue content. A signal is a limited form of inter-thread communication used in POSIX-compliant operating systems. Essentially, it is an asynchronous notification sent to a thread in order to notify it of an event that occurred. When a signal is sent to a thread, the operating system interrupts the thread's normal flow of execution. If the

thread has previously registered a signal handler, that routine is executed. Otherwise, the default signal handler is executed. An AADL event port connection between threads translates into the sending of POSIX signals between threads.

The signals used are the user-definable real-time signals. The structure type of an object used to represent sets of signals must be used with the POSIX functions that initialize and empty a signal set, add a signal to a signal set and examine and change blocked signals before creating the thread. The source/destination POSIX thread that corresponds to the AADL source/destination thread of the event port connection sends/waits for the signal (zero timeout for no blocking if there is no signal received).

Global variables

An AADL data port acts as an interface for typed state data transmission among components without queuing. Data ports are represented by typed variables in source text. A global variable is a variable that is accessible in every scope. Global variables are used extensively to pass information between sections of code that do not share a caller/called relation such as concurrent threads. An AADL data port connection between threads translates into a global variable between threads.

The data type of this global variable is derived from the type of ports connected. The source/destination thread that corresponds to the AADL source/destination thread of the data port connection, can write/read a value in/from that global variable.

The AADL properties are translated as followed:

Scheduling_Policy and Priority of threads. An AADL property set called UC with two properties POSIX_Scheduling_Policy and Priority has been defined. The first is an enumeration of the values SCHED_FIFO, SCHED_RR, SCHED_SPORADIC and SCHED_OTHER, and the second is an integer from 0 to 32. The first is obviously used to set the scheduling policy of the treads. The second is used with the appropriate minimum value for the scheduling policy specified to set the scheduling parameter attributes of the threads.

Compute_Execution_Time (min, max). The minimum time causes a call to a function that consumes that processing time to assure that at least that time is consumed. This function is adjusted at the beginning of the application to assure that the exact time is consumed. Thus the minimum execution time is the time established by this property for this thread.

The maximum time requires the creation of a timer that is set with this time until the next expiration of the timer. Therefore, the timer expires in a maximum time nanoseconds from when the call is made. When this timer expires, one of the last real-time signals is sent and a function called. This function lowers the priority of the thread and waits for a while before restoring the initial priority of the thread using the same method. When the priority of the thread is low, the scheduler avoids executing the thread and other threads can be processed. Thus we assure that the maximum execution time is the one of this property for this thread.

Names. Property Activate_Entrypoint of a thread is the name of the C++ function that contains the source code of that thread. Thus, this is the name of the function executed as a starting routine when creating the thread. Source_Text of a thread is the name of the C++ file containing the source code of that thread.

Initialize / Finalize_Entrypoint. The name of the routine called at the start/end of the start routine of the corresponding thread is derived from this property.

Initialize / Finalize_Execution_Time (min, max). The minimum time causes the call to a function that consumes that processing time to assure that at least that time is consumed. It checks the maximum time, to see if this amount of time has elapsed and return if it has been.

The issues related to the subprograms are the following:

Subprogram. An AADL subprogram component abstraction represents sequentially executable source text, a callable component, with or without parameters, that operates on data or provides server functions to components that call it. A routine is a portion of code within a larger program, which performs a specific task and is relatively independent of the remaining code. An AADL subprogram translates into a routine.

Subprogram calls. In AADL there are two types of subprogram calls: Call sequences and remote calls. The local call from a thread or from another subprogram within the same thread to a subprogram is made in AADL through the sub-clause call and is translated into direct calls from the thread start routine or from the routine respectively.

The remote client-server call from a subprogram in a thread to another subprogram in another thread is made through the sub-clause call and the property Actual_Subprogram_Call. This remote call translates into a call from one routine to another.

Subprogram parameters. A parameter represents call and return data values or references to data passed into and out of a subprogram, so it can be by value or by reference. In AADL the data values are in or out parameters and references are requires data access. Connections must be established between the ports of the thread (or the subprogram) and the ports of the subprogram. The data type of the AADL out parameter, if any, determines the data type of the routine; if there is no out parameter the type is void. Thus, the AADL parameters translate into parameters of the subprogram by value or reference. The translation permits data exchange among subprograms.

AADL data are managed as follows:

Data type. The AADL data abstraction represents static data and data types. Data component declarations are used to represent: application data types, the substructure of data types via data subcomponents within data implementation declarations and data instances. In general, a data type defines a set of values and the allowable operations on those values. Simple independent AADL data gives rise to a data type. These data types will be used later to define the type

of a global variable, a message, etc. The name of the data type can be inferred from the name of the AADL data. This translation takes into account the property Source_Data_Size. In the case of data types, it specifies the maximum size required to hold a value of an instance of the data type.

Simple Data. A simple AADL data subcomponent of a thread or a process gives rise to a simple global variable. The name and type can be inferred from the name and the AADL data type.

Composite Data. Composite AADL data are data that have one or more subprograms as features and/or one or more datum as subcomponents. These data generate a C++ class of data with its methods and/or member data. The name of the class can be inferred from the name of the AADL data. The names and types of the methods and members can be inferred from the AADL subprograms and data. The composite data subcomponents of a thread or a process give rise to a global variable whose type is that class. The name can be inferred from the name of the AADL data.

The AADL behavioural annex improves the specification of a component's behaviour. AADS parses the AADL model so the annex behaviour_specification sections are translated to an equivalent POSIX model.

The behavioural annex describes a transition system (an extended automaton) using optional sections:

State variables. The state variables section declares typed identifiers. Types are data classifiers of the AADL model. AADS translates these state variables declaring variables with their corresponding type in the C++ source code of the thread or subprogram itself.

Initialization. The state variables must be initialized in the initialization section using a sequence of assignments. AADS translates this initialization by initializing the variables with their corresponding value where they were declared.

States

The states section declares automaton states which can be qualified as initial, complete, return, urgent or composite. AADS uses this section to know which states have been defined.

Transitions

The transitions section defines system transitions from a source state to a destination state. The transition can be guarded with events or Boolean conditions. An action part can be attached to the transition. It can perform subprogram calls, message sending or assignments. AADS translates the transitions section into switch and case statements to transit from one state to another. It starts in the initial state and moves to the next state when the guard of the transition is true. Thus the guard of the transition translated by AADS acts as a condition to execute the sentence/s of the state and to change the state. This sentence/s is the action of the transition translated by AADS. If there is no guard there is no condition to check. The guard can be an expression as simple as on i < 5, so AADS will translate it directly.

Depending on the content of the guard and the action of the transition, AADS translates them into the corresponding sentences of source code:

Sending / receiving messages. Messages are sent / received through event or event data ports. If p is an input port: p? de-queues an event port variable, p?x de-queues a datum on an event data port in the variable x. If p is an output port: p! calls Raise_Event on an event port, p!d writes data d in the event data port and calls Raise_Event.

In the first case the guard of a transition is p1?x (where p1 is an in event data port) and the action of that transition is p2!(x+1) (where p2 is an out event data port). AADS translates this case, checking whether a variable arrives at the POSIX message queue associated with port p1. Then the variable is sent through the POSIX message queue associated with port p2, in this case after adding 1 to it.

In the second case the guard of a transition is p1? (p1 is an in event port) and the action of that transition is p2! (p2 is an out event port). AADS translates this case, checking whether the corresponding POSIX signal associated with port p1 has been received. Then the corresponding POSIX signal associated with port p2 is sent.

Subprograms. A behaviour expressed by the annex can be attached to a subprogram implementation. The behaviour can refer to the subprogram parameters and to variables. The automaton specifying the subprogram implementation has one or more return states indicating the return to the caller. While the AADL control flows define the call sequences produced by a subprogram, the annex enables the expression of dependencies between the control flows and state variables or parameters. A subprogram specification can express other calls or notification of events.

In the first case the guard of a transition is p1? (p1 is an in event port) and the action of that transition is subp! (subp is a subprogram). AADS translates this case checking whether the corresponding POSIX signal associated with port p1 has been received. If the signal has been received then the corresponding previously defined subprogram is called.

Parameters can be passed to called subprograms. The action of that transition could be subp!(5->x,2->y) where x and y are two in parameters of the subprogram subp. Then AADS translates it into a call to the subprogram with those two parameters as subp(5,2).

Using the AADL behavioural annex, it is possible to indicate in the action of a transition that the out parameter of a subprogram is the in parameter modified in some way. It could be po!(pi+1), where po is the out parameter and pi the in parameter. AADS translates this case, creating the source code in the subprogram that sums one to the in parameter and assigns the result to the out parameter.

In the last case the guard of a transition is on pi (pi is an in parameter of a subprogram) and the action of that transition is a call to a standard function such as std::cout!. To translate this transition AADS generates the C++ source code that checks whether the in parameter is true and, if it is, calls the standard function cout.

Control structures. Control structures support conditional execution of alternative actions (if, else, end if), conditional repetition of actions (while), and application of actions over all elements of a data component array, port queue content, or integer range (for). The For structure represents an ordered iteration over all elements. Within for structures the element can be referenced by element_variable_identifier, which acts as a local variable with the name scope of for structure.

In the case that the action of a transition contains a conditional structure of the type: if (logical value expression) behaviour_actions[else behaviour_actions] end if, AADS translates it producing the source code with the analogous if else structure in C++, adapting the differences between them.

The same can be said about for and while structures of the type: for (element variable identifier in values) {behaviour_actions} and while (logical value expression) {behaviour_actions}. AADS translates them producing the source code with the analogous for and while structure in C++, adapting the differences between them.

Arrays. To declare collections of data which are considered to be ordered the notion of multiplicity is used. AADS translates multiplicity into a C++ array of data. The type of the array is the same in both AADL and C++.

The HW architecture is structured through the XML file generated by AADS. It is used as part of the configuration parameters of SCoPE and is divided into: HW_Platform, SW_Platform and Application.

HW_Platform. Any AADL implementation of a processor, memory, bus or device must be specified with its category and name in the HW_Components subsection of HW_Platform. The AADL property Assign_Byte_Time is used to set the frequency parameter in the XML file. For memories we use the properties Read_Time and Write_Time. These properties have their values in time units (ns, ms and so on) and must be transformed into MHz. To know the mem_size of a memory, both Word_Count and Word_Size AADL properties are required. Finally the mem_type of a memory is derived from Memory_Protocol in the AADL model. If the component is a processor, proc_type must be specified.

The HW_Architecture and Computing_groups subsections of HW_Platform are next in the XML file. To know the start_addr of a memory we take the AADL property Base_Address. The component and name are inferred from the AADL model. HW components are grouped by buses as they are connected to them in AADL through the connections bus access and the features required bus access.

SW_Platform. This section has two subsections: SW_Components and SW_Architecture. This section takes into account the buses that are defined to make the equivalent nodes. In this section the operating systems are specified.

Application. This section has two subsections: Functionality and Allocation. Filling the Functionality section is straightforward from the AADL model using the property of a thread Activate_Entrypoint for the function and Source_Text for the file. The name is the same as the one of the thread. For the Allocation section we need to know the property of a thread Actual_Processor_Binding, and find out which bus the processor is bound to and then find out which node that bus corresponds to. The AADL name of the thread is used for the name and the component.

6. AADS-T

AADS-T is a version of AADS that admits AADL models that include the AADL Behavioural Annex and generates a source code compatible with the Ravenscar Computational Model.

The real-time behaviour specification of ASSERT models is based on the RCM, a model of concurrency for high-integrity systems that enables formal analysis of the temporal properties of a system using response-time analysis techniques. The model includes a static set of concurrent threads of execution, communicating by means of shared protected data with mutually exclusive read and write access, and a restricted form of conditional synchronization. The model is simple enough to be implemented by a simple, small-size real-time kernel, thus easing the way to the eventual certification of real-time systems based on it.

Twelve properties must be fulfilled to be RCM-compliant; the source code generated by AADS-T obeys all of them. These properties are stated in an internal document of the project titled R1-4 Evaluation of Compliance with the ASSERT Process, written by J. A. de la Puente and J. Zamorano.

Basic elements. There are two main elements in the RCM: Threads and protected objects (PO). A thread is the basic unit of execution, which can be executed concurrently with other threads on a single processor. POs are an abstraction of shared data, synchronization, and interrupt handling.

There are a static number of threads and POs. Therefore, threads and POs can only be created at system initialization time.

RCM 1 A real-time system consists of: A static set of N threads, $T = \{t_i\}$, $i \in 1..N$; and a static set of M POs, $O = \{q_i\}$, $i \in 1..M$. The set O may be empty (M = 0), in which case the system is said to have only independent threads.

In the source code generated by AADS-T all the threads and POs are created calling *pthread_create* and as objects of the corresponding classes respectively at system initialization time.

Properties of threads. A thread is a concurrent unit of execution with the following properties:

RCM 2 Threads are non-terminating. They exhibit an endless repetitive behaviour, alternating between the following states (see Figure 5): Suspended (a suspended

thread is not eligible for execution) and Ready (a ready task can be executed when the processor is allocated to it).

Figure 5. States of RCM threads

RCM 3 Threads have a single activation point. An activation point is a point in the executable code of a thread at which its state changes from Suspended to Ready. When activated, a thread becomes ready and then executes a piece of sequential code (thread activity), after which it becomes suspended awaiting the next activation.

Threads of the source code generated by AADS-T use *while(true)* to be non-terminating. They are suspended after executing the sequential code in a *clock_nanosleep* and when sleeping time has passed they become ready at their single activation point.

RCM 4 The activity of a thread is a sequence of code with a bounded and known worst-case execution time (WCET). The WCET of thread t_i is C_i.

AADS-T utilizes the AADL property *Compute_Execution_Time* to know the WCET of a thread. The source code generated checks that this WCET is not exceeded.

This property implies that a thread does not execute any operation that could result in its becoming suspended other than the suspension immediately before the activation point, and nether do the threads created by AADS-T.

RCM 5 A thread can be activated only by one of the following two kinds of events. One is by a timing event which is issued periodically by the environment. In this case the thread t_i is said to be periodic or time-driven with period T_i.

The other is a synchronization event issued when the barrier of a synchronization PO is opened (see RCM 8 below). In this case, the thread t_i is said to be sporadic. The synchronization event must have a minimum inter-arrival time associated to it, i.e. a minimum elapsed time interval between two consecutive occurrences of the event, T_i.

AADS-T uses the AADL properties *Period* and *Device_Dispatch_Protocol* to know the period and the type of a thread respectively. It accepts only *periodic* and *sporadic* threads and not *aperiodic* or *background* threads. The difference between the codes generated is that a sporadic thread waits for an event from an *event* or *eventdata port connection* after invoking a synchronization operation in the activation point. In both cases *clock_nanosleep* waits a time T_i.

Properties of protected objects. A PO is an object which encapsulates a set of data and a set of associated operations (protected operations). The value of the data makes up the state of the object. The state can only be read or changed by invoking one of the operations of the PO. If q is a PO: $q.S$ denotes its state, $q.S \in \mathbf{S}$, where **S** is an appropriate data domain; $q.P_k$ denotes the k-th operation of q. Notice that a PO must have at least one operation; otherwise its state is inaccessible. The notation $t \rightarrow q$ will be used to denote that t invokes one or more operations of q. Similarly, $t \rightarrow q.P$ means that t calls the operation $q.P$.

AADS-T generates an object of the corresponding class which is a PO in the source code for each AADL *data, event and eventdata port connection*. Classes generated by AADS-T have the appropriate data members to achieve the communication of data and/or events between threads. Each class has a constructor and member functions read and write to initialize and access data members.

POs have the following properties:

RCM 6 Only one thread can be executing an operation of a given PO at any given time, i.e. protected operations are mutually exclusive. Consequently, if a thread invokes a protected operation at a time when another thread is already executing an operation of the same object, it has to wait. When the protected operation that was being executed is completed, the waiting thread is allowed to execute the operation it had invoked. Notice that a thread that is waiting to begin a protected operation is not considered to be suspended. In consequence, a thread activity can invoke protected operations without violating RCM 4.

Each class produced by AADS-T defines a *mutex* that is locked when a member function is called and unlocked when it ends, ensuring compliance with mutual exclusion.

RCM 7 All protected operations have a bounded and known WCET. The WCET of the protected operation $q_i.P_k$ is $C_{i,k}$. Again, this property implies that no operations that could result in a thread being suspended can be invoked from a protected operation.

AADS-T uses the ad hoc defined AADL properties *PO_read_WCET* and *PO_write_WCET* for each *port connection* to know the WCET of each member function. The source code generated checks if these WCETs are exceeded. Moreover, no member function calls any suspending operation.

RCM 8 A PO can have at most one synchronization operation that has an associated barrier, which is a Boolean variable that is part of the object state. When the value of the barrier is true, the barrier is said to be open, and otherwise it is said to be closed.

The behaviour associated with synchronized operations is as follows: When a thread invokes a synchronization operation, if the barrier is open the execution proceeds as with an ordinary protected operation; but if the barrier is closed, the thread is suspended. At most one thread can be

suspended at a barrier at any given time. A thread that is suspended at a barrier is resumed whenever the barrier becomes true (as the result of the execution of another protected operation by some other thread).

Invoking a synchronization operation is a potentially suspending operation, and thus cannot be done within a thread activity; this can only be used to implement the activation events of sporadic threads.

In the source code produced by AADS-T only the objects corresponding to *event* and *eventdata port connections* have a synchronization member function because a sporadic thread is dispatched by an event as stated above. Only sporadic threads invoke the synchronization. The barrier is initialized as false in the constructor, then set to true in the write member function, then checked to see whether it is false in the synchronization to suspend the thread on a *nanosleep*, and finally set to false after resuming it.

Scheduling. The RCM is associated with an instance of the fixed-priority pre-emptive scheduling (FPPS) method, together with the immediate ceiling priority inheritance protocol (ICPP). The scheduling model is defined by the following properties:

RCM 9 Each thread t_i has a basic priority, $P_i \in \mathbf{P} \subset Z$, where Z is the set of the integer numbers. The basic priority of a thread is fixed, i.e. it is never changed.

AADS-T uses the ad hoc AADL property *Priority* to create a thread at system initialization time with *sched_priority* at that priority, which is never changed.

RCM 10 Each PO q_i has a ceiling priority CP_i which is the maximum of the basic priorities of all the threads invoking any of its operations: $CP_i = \max\{P_j, t_j \rightarrow q_i\}$. As basic priorities of all threads are fixed, so too are the ceiling priorities of all POs.

RCM 11 At every instant of time, each thread has an active priority. The active priority of a thread is the maximum of the basic priority of the thread and the ceiling priority of all POs that contain an operation that is currently being executed by the thread. Therefore, whenever a thread invokes a protected operation, it immediately inherits the ceiling priority of the enclosing PO.

In the source code generated by AADS-T the function *pthread_mutexattr_setprotocol* is used with the value *PTHREAD_PRIO_PROTECT* and the function *pthread_mutexattr_setprioceiling* with the maximum of the priorities of the two threads communicating through a *port connection*. This is done when initializing the *mutex* of the object corresponding to that *connection* at system initialization time guaranteeing the fulfillment of RCM 10 and RCM 11.

RCM 12 Ready threads are conceptually grouped into ready queues. There is a ready queue for each priority level in **P**. Threads are added to and removed from priority queues according to the following rules: When a suspended thread becomes ready, it is added at the tail of the priority queue for its active priority. When the processor is idle, the thread which is at the head of the non-empty ready queue with the highest priority is dispatched for execution and removed from the queue. Whenever there is a non-empty ready queue with a higher priority than the priority of the currently running thread, the thread is pre-empted from the processor and it is added at the head of the ready queue for its active priority. Notice that according to the previous rule, the thread at the head of the ready queue that caused the pre-emption is dispatched for execution immediately afterwards.

AADS-T admits only *SCHED_FIFO* for the ad hoc AADL property *POSIX_Scheduling_Policy* of a thread, to set so *sched_policy* in the source code.

The above model specifies a concurrent system with a predictable, analyzable temporal behaviour. Since the execution time of threads is bounded (RCM 4, RCM 7) and the scheduling method is FPPS with ICPP, well-known response-time analysis techniques can be applied to statically guarantee that the system satisfies its temporal requirements.

7. Experimental Results

7.1. Case Study

The proposed method implemented in AADS-T was tested assisting in the HW/SW partitioning of the case study shown in Figure 6. It is a space application of digital image processing that consists of different components: HW Controller, Image Processing, Image Filter, Control, Start-up, Housekeeping, Monitoring, Image Processing Management, Event Generation, Connection test and Stub Simulating Ground. HW Controller is in charge of receiving images (bitmap images) from the Camera Simulator as well as sending the filtered images back to Ground. Image Processing receives the image from HW Controller pixel by pixel, including the padding when it is needed, forwarding them to Image Filter to process them. Image Processing also receives the pixels already filtered by Image Filter which are subsequently sent to the HW Controller component. Image Filter is in charge of filtering the pixels received according to the filter selected. Control manages the Image Application Software based on OBSW commands in order to store image statistics, to configure image filtering function and to control the image processing status. Start-up simulates the initialization routine and starts the next OBSW functionalities. Housekeeping periodically reports the number of images correctly filtered. Monitoring checks the status of the Image Processing Application Software and if the status is set to erroneous, an event is sent to Stub Simulating Ground through Event Generation. Image Processing Management selects the filter to be applied and starts/stops the application. Event Generation simulates the OBSW event reporting service to inform ground about all asynchronous events occurring on-board. Connection test

lets operators test the OBSW application presence and state. Stub Simulating Ground requests the OBSW to perform the alive-test by sending the corresponding telecommand.

Figure 7 is the AADL graphical notation of the Case study with the memories (two DRAMs, one for the FPGA and another for the LEON2), processors (FPGA and LEON2), buses (RS232, FPGA memory bus, and LEON2 memory bus), threads, event ports and eventdata ports generated with OSATE v1.5.8.

AADL provides many benefits for HW/SW co-design. It contains constructs for modeling both HW and SW components. This language supports early and repeated analyses of system architecture with respect to performance-critical properties through an extendable notation, a tool framework and precisely defined semantics.

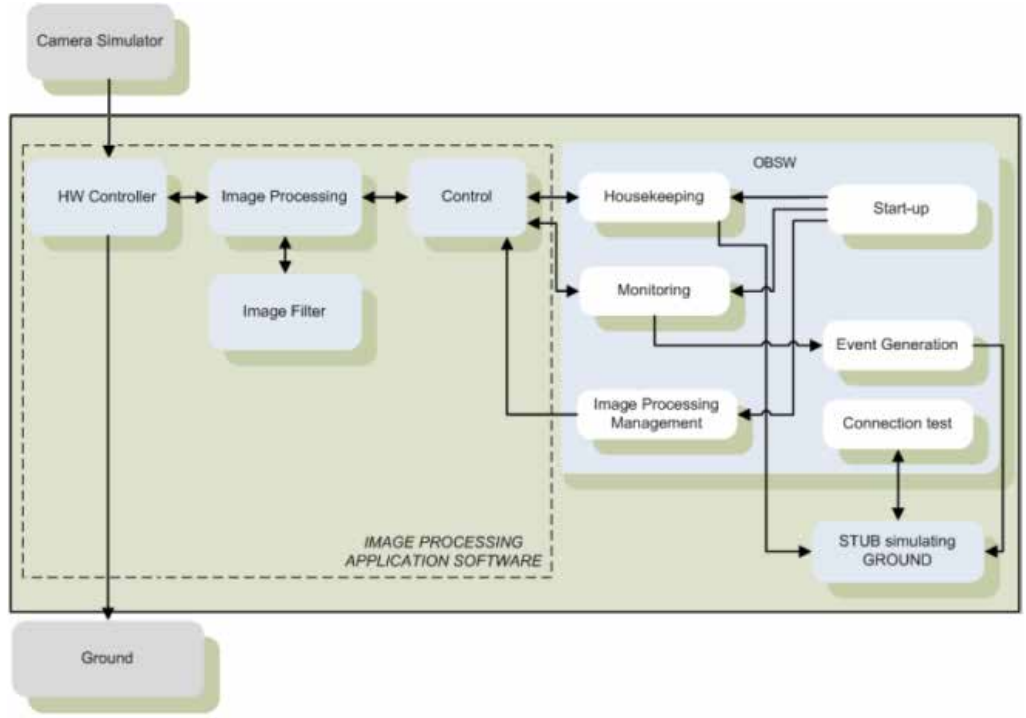

Figure 6. Case study functional description

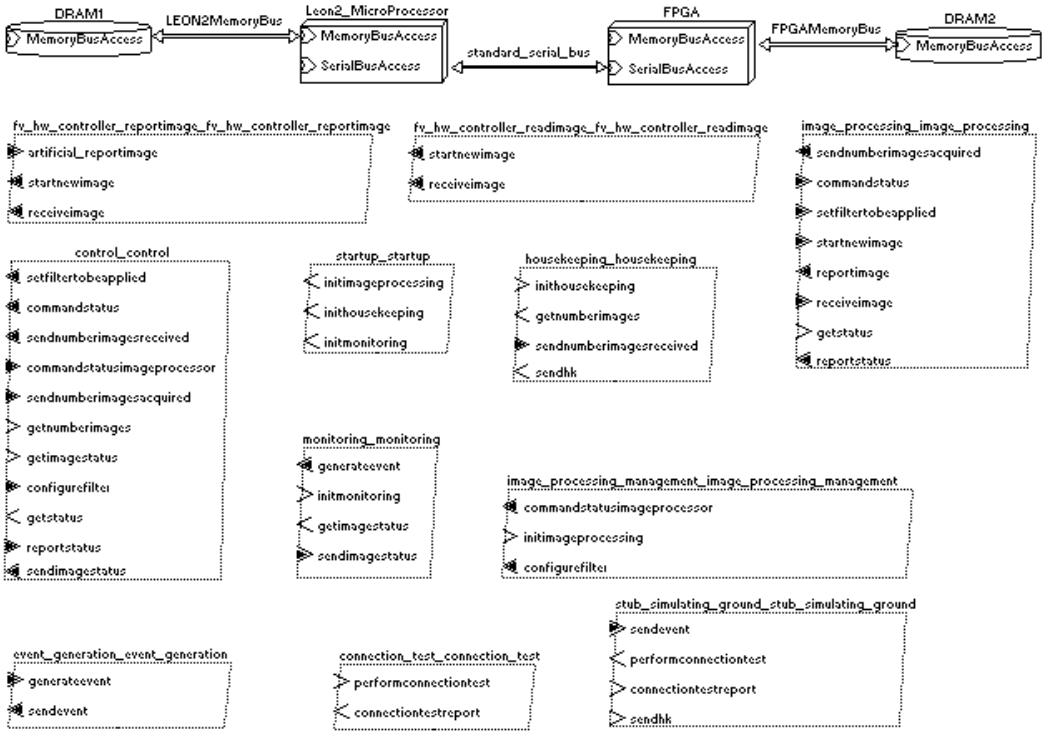

Figure 7. AADL graphical notation of the Case study

AADS extracts from the AADL models the necessary information for the SCoPE tool to perform a simulation at system level. The simulation results will guide the system designer through the selection of the most adequate partition solution.

7.2. LEON2 Modelling in SCoPE

SCoPE has been modified to include the LEON2 processor at 50 Mhz, 15.4 MIPS and 30.64 nJ of energy consumed per instruction in its *processors.xml* configuration file. It was necessary to specify the data and instruction cache sizes too. A *size* of 8192, a *size of line* of 8 and an *associativity* of 1, considering an *instruction size* of 4 (32 bits), was considered for both. Another configuration file of SCoPE, *meminst.xml*, was modified to include the operation codes of the LEON2.

The GNU cross-compiler for LEON2 used is the GNAT for the LEON 2.1.0 C compiler so the source code produced by AADS-T has to comply with certain characteristics. The options *POSIX_THREADS, POSIX_THREAD_PRIORITY_SCHEDULING, POSIX_THREAD_PRIO_PROTECT, LEON_2* and *POSIX_TIMERS* have to be activated. The function *clock_nanosleep* must be explicitly declared.

7.3. Architectural Design

The partitioning process is performed following a SW centric approach. Firstly, it is assumed that all system functions are SW components. If this assumption is not fulfilled due to the violation of the design criteria, new partitions are proposed in order to accommodate system functions to other platform resources. In this case, different allocations of system functions to platform resources are possible. The decision about which parts are mapped to HW is based on the analysis of which implementation best meets the design criteria (derived from the requirement analysis results) in terms of performance and functional behavior. The information about how a component will be implemented (HW or SW) is added as an AADL property, Actual_Processor_Binding. The partitioning is generated by the ASSERT Model Transformation (AMT) tool developed by GMV.

In the context of a SW centric approach, initially all system functions will be mapped onto the same processing node, whose implementation is a LEON2 microprocessor. We considered a situation where it is necessary to re-allocate system functions from SW to HW (i.e. mapping from processing elements whose technological implementation is a microprocessor to other processing elements such as DSP, PLD, FPGA, etc): The system performance might indicate that the CPU exceeds the maximum performance limits imposed in the requirements.

HW/SW models were generated in AADL by AMT and in SystemC with AADS-T/SCoPE, to perform the HW/SW partitioning. The system-level performance tool (AADS-T/SCoPE) was used to analyze the system performance and non-functional requirements such as use of CPU, timing or energy consumption for a given HW/SW partition. After system performance analysis, some HW/SW partitions were proposed and evaluated.

The files produced by AADS-T were compiled with SCoPE to simulate the model and the results obtained were used to compare the different partitions. The simulation executed the source code of the threads and the protected objects enabled communication among the threads.

In the following table we can see the comparison between the sixteen magnitudes evaluated by SCoPE in the average of the simulations carried out with fifty-six evaluated partitions, and the one which allocates the components Control and Image_processing to HW. In nearly all the magnitudes the selected partition obtains the best performance results.

Table 1. Comparison between simulations' metrics of the selected partition and the average of the others

	Control &	Average of other
Total User time	0.424528 s	2.2550830 s
Total Kernel	0.00550586 s	0.0314650 s
Number of	20172	34717
Running time	382438640 ns	2026357221 ns
Use of CPU	0.382439 %	2.0263575 %
Instructions	6016438	34230189
Instruction cache	849919	4646324
Core Energy	1.84344e+08 nJ	1.04881E+09 nJ
Core Power	1.84344 mW	10.48813 mW
Instruction cache	7.97346E+08 nJ	4.35719E+09 nJ
Instruction cache	7.97346 mW	44.20486 mW
Bus access time	47595464 ns	260191004 ns
Idle time	99559965896 ns	97703658860 ns
Number of	12550	11119
Instruction miss	10116	51850
Bus Load	27197408 bytes	148680573 bytes

In the following figure we can see one magnitude (Use of CPU from a total of sixteen magnitudes evaluated by SCoPE) of the simulations carried out with SCoPE with the fifty-seven evaluated partitions. Among all the partitions, one obtains the best performance results and so it is selected; the one that allocates to HW the components Control and Image Processing because they consume most of the resources. In the figure first we can see the partitions that allocate one component to HW, then we can see the partitions that allocate two or more components to HW. Allocating only one component to HW was not enough to fulfil the initial constraint as we will see below, so more than one component had to be allocated to HW. However we had to take into account that allocating to HW is more expensive than allocating to SW so we could not allocate all the components to HW. We had to maintain a trade-off between the cost of allocating all the components to HW and the use of CPU caused by allocating the components to SW.

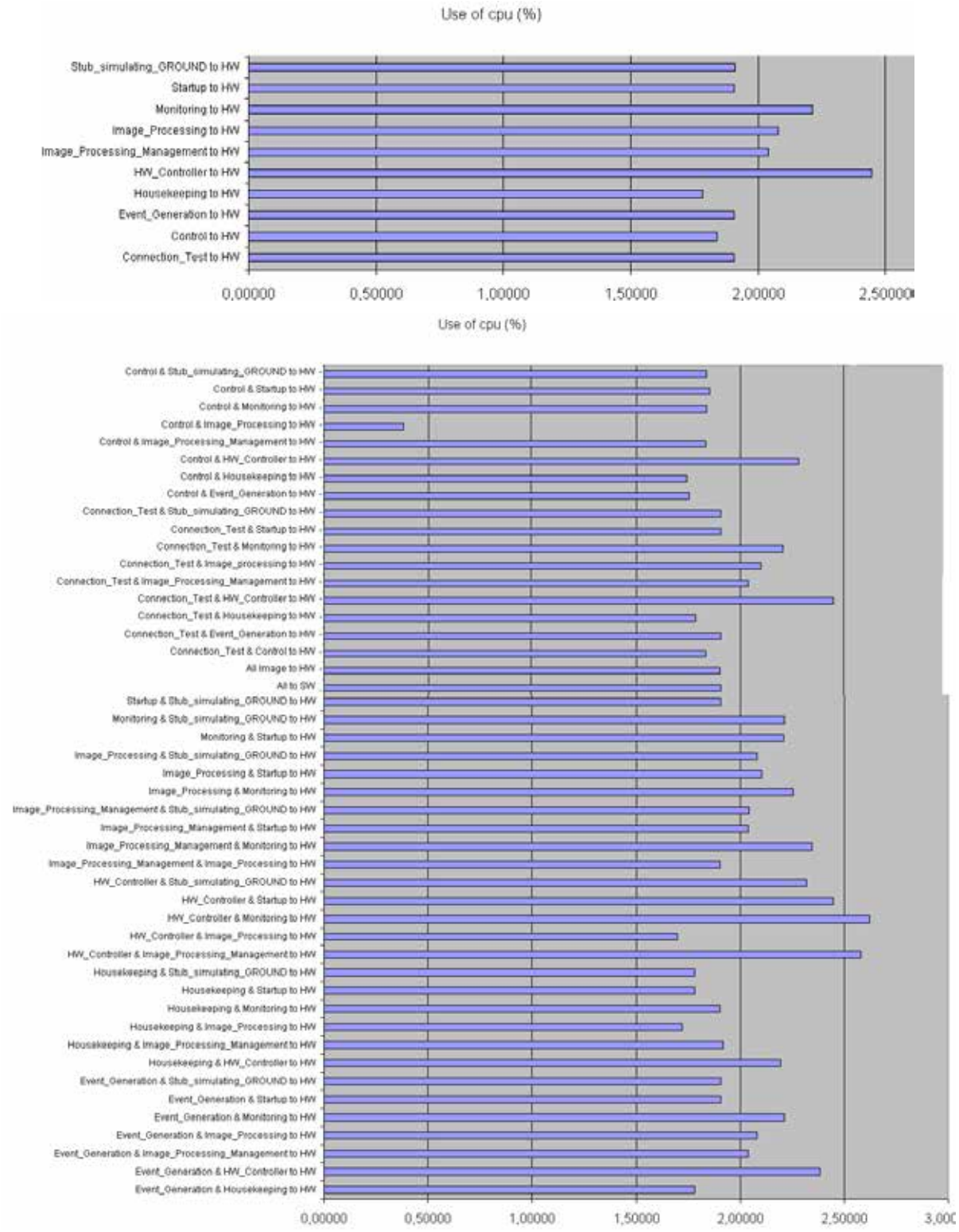

Figure 8. Use of CPU (%) of the partitions

In order to check how the performance analysis results vary depending on the WCET, the AADL model was manually modified setting very strict WCET (close to the deadlines). In this case study the deadlines of the threads were 20 % of the periods. So the WCET were 20 % of the values from the interval of 40 ms to 100 s which were the minimum and the maximum periods respectively for the different threads. Regarding only the Use of CPU, the initial partition which considers that all components were allocated to SW consumed 1.90599 % with the initial WCET values. However with the WCET close to the deadlines, the Use of CPU had a significant increase up to

97.6006 %. The selected partition that allocates the components Control and Image Processing to HW, consumed 0.382439 % with the initial WCET values. With the WCET close to the deadlines, the Use of CPU was only 14.32543 %. As the components which consume most of the resources had been moved to HW, the processor provided a completely different result. When all components were mapped to SW, the usage of CPU was 97.6 %. As CPU load requires a workload margin, this scheme is not valid. Therefore, partitioning is required.

The initial constrain imposed on the system was a Use of CPU less than 75 % in the worst case. This worst case is when the WCET of the threads were close to the deadlines. In this case the only HW/SW partition that complies with the initial constraint is selected, namely the one that allocates the components Control and Image Processing to HW.

Allocating the components Control and Image Processing to HW leads to a significant reduction in use of CPU due to the improvement in the communications between the two components. This shows that performance analysis enables the discovery of situations that would be difficult to expose without such a powerful tool.

8. Conclusions

This document describes the simulation of AADL compatible with RCM using the AADS simulation tool. AADS supports the refinement of AADL models, including the Behavioural Annex, through performance analysis done with SCoPE, after translating those models.

The generation of the RCM-compliant SystemC model from the AADL specification is not straightforward. Nevertheless, the SystemC model generated by AADS is able to capture the fundamental dynamic properties of the initial system specification. In this way, AADS supports design space exploration by refinement of the AADL functionality and its implementation on an optimized platform.

The system-level performance tool (AADS-T/SCoPE) is used to analyze the system performance and non-functional requirements such as use of CPU, timing or energy consumption for a given HW/SW partition. AADS-T aids the performance of the HW/SW partitioning of a system by analyzing HW/SW models.

Future work includes incorporation of AADS features that appear in V2.0 of the AADL standard.

ACKNOWLEDGEMENTS

This work has been developed in the University of Cantabria and has been partially supported by the Spanish MICyT through the ITEA 05015 SPICES project[47], the TEC2008-04107 project, and the ESTEC 22810/09/NL/JK HW-SW CODESIGN project contracted to GMV Aerospace and Defence S.A.U.

The authors would like to acknowledge the aid received in this work from Juan Antonio de la Puente and Juan Zamorano of UPM, as well as from the colleagues in the SPICES project and in the Microelectronics Engineering Group (GIM).

REFERENCES

[1] SAE: AADL. June 2006, document AS5506/1. www.sae.org/technical/standards/AS5506/1.

[2] P. H. Feiler, D. P. Gluch, J. J. Hudak: The AADL: An Introduction. CMU. Pittsburgh. (2006).

[3] P. H. Feiler, J. J. Hudak: Developing AADL Models for Control Systems: Practitioner's Guide. CMU. 2006.

[4] SAE. Annex Behavior V1.6 AS5506, 2007.

[5] www.assert-project.net 2008 ESA/ESTEC.

[6] A. Burns et al.: The Ravenscar tasking profile for high integrity RT programs. Ada-Europe'98. Springer-Verlag.

[7] M. Perrotin et al.: The TASTE toolset: Turning human designed heterogeneous systems into computer built homogeneous software. ERTS2 2010, Toulouse, France.

[8] LEON2-FT ESA Microelectronics 2009 www.esa.int/TEC/Microelectronics/SEMUD70CYTE_0.html

[9] A.D. Pimentel et al.: A systematic approach to exploring embedded system architectures at multiple abstraction levels, IEEE Transactions on Computers, 2006.

[10] J. Hugues, B. Zalila, L. Pautet, F. Kordon: From the prototype to the final embedded system using the Ocarina AADL tool suite. ACM TECS, 2008. NY, USA.

[11] H. Posadas et al.: RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model. Design Automation for Embedded Systems. Springer. 2005.

[12] AADS UC 2011. www.teisa.unican.es/AADS

[13] P. H. Feiler, A. Greenhouse: OSATE Plug-in Development Guide. CMU. Pittsburgh. (2006).

[14] The Eclipse Foundation 2009. www.eclipse.org

[15] J.-F. Tilman, R. Sezestre, A. Schyn: Simulation of system architectures with AADL. ERTS2008, Toulouse.

[16] F. Singhoff, A. Plantec: AADL modeling and analysis of hierarchical schedulers. SIGAda'07, Fairfax, VA, USA.

[17] O. Sokolsky, I. Lee, D. Clark: Schedulability Analysis of AADL models. IPDPS 2006. Rhodes Island, Greece.

[18] M. Yassin Chkouri, A. Robert, M. Bozga, J. Sifakis: Translating AADL into BIP – Application of Real-time Systems. ACESMB 2008. Toulouse, France.

[19] A. E. Rugina et al.: The ADAPT tool: From AADL architectural models to stochastic Petri Nets through model transformation. EDCC. 2008. Kaunas, Lithuania.

[20] T. Abdoul, J. Champeau, P. Dhaussy, P. Y. Pillain, J. C. Roger : AADL execution semantics transformation for formal verification. ICECCS 2008. Belfast, U. K.

[21] E. Jahier et al.: Virtual execution of AADL models via a translation into synchronous programs. EMSOFT'07. 2007. Salzburg, Austria.

[22] S. Gui et al.: Formal schedulability analysis and simulation for AADL. ICESS2008. Chengdu, China.

[23] M. Brun, J. Delatour, Y. Trinquet: Code generation from AADL to a RTOS: an experimentation feedback on the use of model transformation. ICECCS. 2008. U. K.

[24] SCoPE UC 2011. www.teisa.unican.es/scope

[25] David C. Black, Jack Donovan: SystemC: From the ground up. Kluwer Academic Publishers. Boston (2004).

[26] M. González: POSIX tiempo real. UC, Santander 2004.

[27] The Open Group: The Single UNIX Specification, V. 2, 1997. www.opengroup.org/onlinepubs/007908799.

[28] W3C: Extensible Markup Language (XML) W3C Recommendation (2006). www.w3.org/TR/REC-xml/

[29] P. Dissaux, J. P. Bodeveix, M. Filali, P. Gaufillet, F. Vernadat: AADL behavioural annex. DASIA 2006. Berlin.

[30] R. Bedin, J. P. Bodeveix, M. Filali, J. F. Rolland, D. Chemouil, D. Thomas: The AADL behavior annex – experiments and roadmap. ICECCS 2007. New Zealand.

[31] J. P. Bodeveix, M. Filali, M. Rached, D. Chemouil, P. Gaufillet: Experimenting an AADL behavioural annex and a verification method. DASIA 2006. Berlin, Germany.

[32] B. Berthomieu, J. P. Bodeveix, C. Chaudet, S. Dal Zilio, M. Filali, F. Vernadat: Formal Verification of AADL Specifications in the Topcased Environment. Ada-Europe 2009. Brest, France.

[33] C. Ponsard, M. Delehaye: Towards a model-driven approach for mapping requirements on AADL architectures. ICECCS 2009. Potsdam, Germany.

[34] M. Yassin Chkouri, A. Robert, M. Bozga, J. Sifakis: Translating AADL into BIP – Application of Real-time Systems. ACESMB 2008. Toulouse, France.

[35] I. Malavolta, H. Muccini, P. Pelliccione: Integrating AADL within a multi-domain modelling framework. ICECCS 2009. Potsdam, Germany.

[36] J. A. de la Puente et al.: The ASSERT VM: A Predictable Platform for Real-Time Systems. IFAC08. Korea.

[37] J. Zamorano et al.: The ASSERT VM kernel: Support for preservation of temporal properties. DASIA 2008. Spain.

[38] M. Bordin et al.: Automated Model-based Generation of Ravenscar-compliant Source Code. ECRTS05. Spain.

[39] S. Mazzini et al.: An MDE Methodology for the Development of High-Integrity RT Systems. DATE09. Nice.

[40] J. Kwon et al.: Ravenscar-Java: a High-Integrity Profile for Real-Time Java. ACM-ISCOPE, 2002. Washington.

[41] SAX: Simple API for XML. (2004). www.saxproject.org

[42] R. Varona Gómez, E. Villar: AADL Simulation and Performance Analysis in SystemC. ICECCS 2009. Germany.

[43] R. Varona Gómez, E. Villar: AADS+: AADL Simulation including the Behavioral Annex. ICECCS 2010. Oxford.

[44] R. Varona Gómez, E. Villar, A. Rodríguez: Ravenscar Computational Model compliant AADL Simulation on LEON2. ISSE 2011. Orlando.

[45] Hardware/Software Codesign Overview RASSP Education & Facilitation Program Module 14, RASSP, 1999.

[46] http://hwswcodesign.gmv.com

[47] http://www.spices-itea.org

[48] R. Bedin, J. F. Rolland, M. Filali, J. P. Bodeveix: Assessment of the AADL Behavioral Annex. FAC2007. Toulouse.

[49] C. Silvano, W. Fornaciari, E. Villar: Multi-objective Design Space Exploration of Multiprocessor SoC Architectures. Springer 2011.

[50] M.B. Abdelhalim, S.E.-D. Habib: An integrated high-level hardware/software partitioning methodology. Design Automation for Embedded Systems. Springer 2011.

[51] K. Lampka, S. Perathoner, L. Thiele: Analytic real-time analysis and timed automata: a hybrid methodology for the performance analysis of embedded real-time systems. Design Automation for Embedded Systems. Springer 2010.