CHESS

Composition with Guarantees for High-integrity Embedded Software Components Assembly

Project Number 216682

D4.3 – Predictability property-preservation needs

Version 1.1

26 January 2011

Final

Public Distribution

UPD, UPM, FhG, Atego, Aicas, MDH, TCF, ENEA

Project Partners: **Aicas, Atego, Atos Origin, CNRI-ISTI, Enea, Ericsson, Fraunhofer, FZI, GMV Aerospace & Defence, INRIA, Intecs, Italcertifer, Maelardalens University, Thales Alenia Space, Thales Communications, The Open Group, University of Padova , University Polytechnic of Madrid**

Every effort has been made to ensure that all statements and information contained herein are accurate, however the Partners accept no liability for any error or omission in the same.

© 2009 Copyright in this document remains vested in the CHESS Project Partners.

DOCUMENT CONTROL

Version	Status	Date
0.1	Table of contents according to comments from Brussels meeting 19 May 2010	25 May 2010
0.2	Initial contents for scheduling analysis from UPD and UPM	25 July 2010
0.3	General modifications	31 August 2010
0.5	Initial contents for simulation analysis from FhG	5 November 2010
0.6	Some modification of UPM in section 3	30 November 2010
0.7	Introduced table of requirements in appendix	21 December 2010
0.8	End-to-end review of document (UPD)	22 December 2010
0.9	Removal of redundancies in the specification of the Ravenscar profile	23 December 2010
1.0	Version before internal review, properties for Linux	20 January 2011
1.1	Modifications after review	26 January 2011

TABLE OF CONTENTS

1. Introduction **5**

1.1 Description of the Problem *5*

1.2 Some Examples of Run-Time Semantic Inconsistency *7*
1.2.1 Parameter direction and concurrent semantics 7
1.2.2 List of consistency requirements for the Ravenscar Computational Model 8

2. General solutions to ensure consistency between models and execution platforms **9**

2.1 Model specification of mappings from analysis patterns to run-time patterns *9*
2.1.1 Model specification of mappings from schedulability analysis to violation events 9
2.1.2 Model Assumptions of Deployment Determination Analysis 10
2.1.3 Assumptions about simulation analysis to run-time patterns 11

2.2 Definition of violation events to preserve run-time consistency with the analysis *12*
2.2.1 Scheduling Analysis: Thread management, synchronization and communication 12
2.2.2 Scheduling Analysis: Time management 14
2.2.3 Scheduling Analysis: Interrupt management 14
2.2.4 Simulation Analysis: Violations of User Model Assumptions 14
2.2.5 Memory management 15

3. Multiplatform consistency specifications **15**

3.1 Scheduling *15*

4. Platform specific consistency specifications **16**

4.1 Ada Ravenscar Profile/LwCCM Platform *16*
4.1.1 Task management, synchronization and communication 16
4.1.2 Time management 17
4.1.3 Interrupt management 18
4.1.4 Memory management 18

4.2 Linux/OSE Platform *18*
4.2.1 Linux 18
4.2.2 OSE 19

4.3 RTSJ Platform *22*
4.3.1 Task management, synchronization and communication 22
4.3.2 Time management 24
4.3.3 Interrupt management 24
4.3.4 Memory management 24

5. References **25**

APPENDIX A. Requirements on platforms for the consistency of analysis methods **29**

FIGURES

Figure 1-1: CHESS tool chain 5

EXECUTIVE SUMMARY

This report introduces solutions adopted in CHESS to make consistent analysis models and execution platforms properties. The objective of this report is to make as much consistent as possible the results of analysis and their equivalent run-time execution temporal properties.

Different analysis methods assume some properties of execution platforms; this deliverable includes these assumptions and some solutions to make analysis and execution consistent. Specific platforms can address the assumptions with different solutions. The analysis methods try to be as much platform independent as possible, but their application in specific platform requires some specific customizations and can have particular restrictions.

1. INTRODUCTION

In the CHESS tool chain several modelling tools and artefacts handle the same information base for different purposes. It is important however that this information be handled and interpreted with the same semantics. Figure 1-1 shows a bird's eye overview of the CHESS tool chain. This deliverable addresses the issue of how to ensure the consistency of the forward and backward transformations to be made from CHESS ML to analysis tools, from implementation languages and platforms to CHESS ML, and the generators from CHESS ML to implementation languages. Code generators, transformers to analysis and code analyzers and transformers to CHESS ML, and execution platforms must be all semantically consistent, for each analysis method.

Analysis methods are designed and implemented to handle specialized problems, and to be applicable in practice they make specific assumptions and impose restrictions and limitations. This deliverable discusses the limitations imposed by analysis languages and tools on generators and platforms in order to make the analysis applicable and trustworthy its results. The analysis methods assume some restrictions on the static and dynamic nature of programs; these restrictions are discussed in this deliverable, to enhance the execution platforms if and where needed to comply with the suite of analysis used in CHESS and to make the code generators preserve the form required of their source code products.

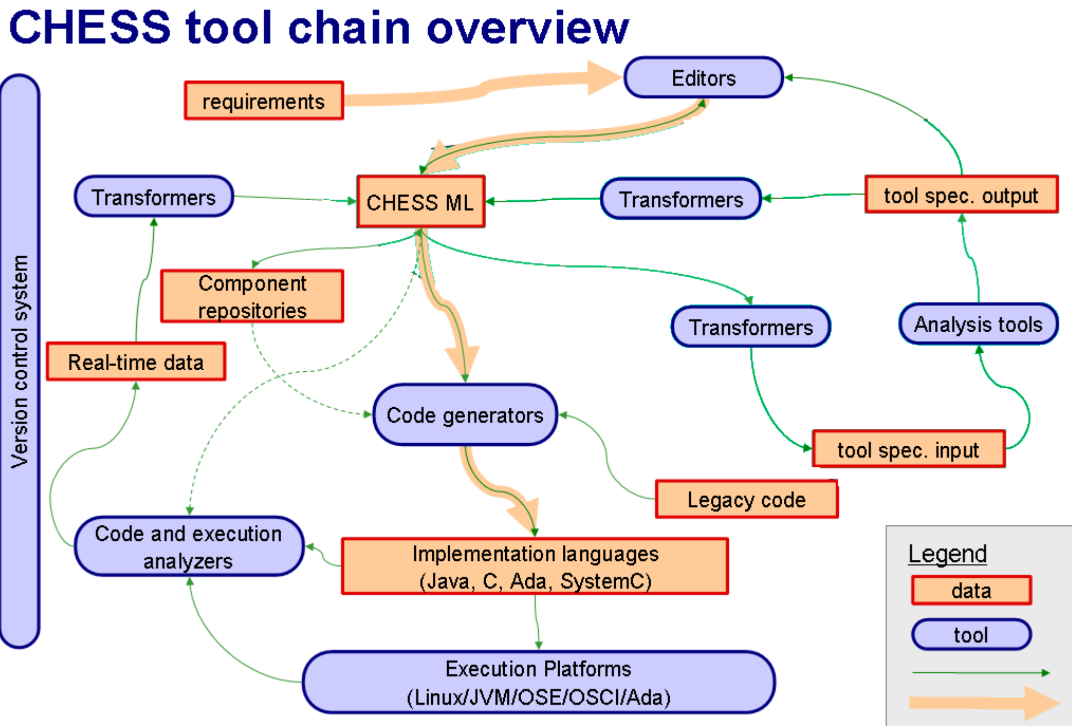

Figure 1-1: CHESS tool chain

1.1 DESCRIPTION OF THE PROBLEM

The four key technical ingredients of the development approach adopted in the CHESS project are: (i) a *component model*, to design reusable software components; (ii) *the computational model*, which describes the allowable semantics and the necessary

constraints to develop analyzable software entities and relates the design entities and their attributes to a set of analysis equations of the underlying analysis body of knowledge; (iii) a *programming model*, as a tailored subset of a chosen set of implementation language which, together with a set of code archetypes, is able to express in the implementation solely and exactly the execution semantics assumed by the analysis theory and to convey in the implementation the realization of the extra-functional attributes used as input for the analysis; and (iv) a conforming *execution platform*, which is in charge of warranting the properties that were asserted by analyze and cope with run-time violations w.r.t. non-functional concerns.

The component model is defined and implemented in WP2. The computational model(s) is/are defined in WP4 in conjunction with the selection of the analysis theory and techniques to be used in the predictability dimension. The programming model(s) and the execution platform(s), which may vary with the industrial domain addressed by the project, are specified and supported in WP5.

The component model we are developing in CHESS is agnostic on the underlying computational model. To this end we strive to maintain the component model void of predefined semantics for what concerns the extra-functional aspects. While the component model of course is equipped with the syntactic means to specify all the extra-functional attributes of interest, only at the point of binding to the computational model of choice those attributes taken a given semantics which fits exactly the space allowed by the chosen analysis theory in the extra-functional dimension of interest.

When the computational model of choice is selected, the software model must then fully abide by all the semantic assumptions and constraints entailed by it. The choice of the computational model must obviously also precede the analysis of the model and thus the automated generation of the Schedulability Analysis Model (SAM) which is input to the analysis engine.

The following three issues must be carefully addressed to make sure that the above vision and notions hold:

1. The model transformation from PIM to PSM in general and to the SAM in particular;

2. The transformation from PSM to system implementation in terms of source code and bindings to the middleware;

3. The execution platform.

As regards issue 1, we must make provably sure that the exercised transformations should not generate PSM/SAM model elements or assemblies that conflict with the semantics allowed by the computational model of choice.

As regards issue 2, the transformation to source code and middleware bindings must conform to the programming model of choice and shall not introduce language constructs and calls that break the semantics assumed and allowed by the analysis theory in use. Moreover, the model-to-code transformations shall create, possibly by use of predefined and proven code archetypes: (i) code structures that support the static enforcement of timing properties; (ii) code structures that permit the monitoring of properties whose enforcement can only be made at execution time.

At the time of this writing, the only computational model that has clearly shown to fit the CHESS needs and vision is the Ravenscar Computational Model (RCM) [54, 55]. In the following section we illustrate some examples of semantics imposed by the RCM and situations in which that semantics is violated.

1.2 SOME EXAMPLES OF RUN-TIME SEMANTIC INCONSISTENCY

1.2.1 Parameter direction and concurrent semantics

Operations declared in provided and required interfaces have a signature that specifies the operation name and an ordered list of parameters and exceptions. Each parameter is typed with a data type whose definition must be accessible to the caller, and has a parameter *direction*. The value for parameters direction can be "*in*" (the actual parameter is only read inside the operation), "*out*" (the actual parameter is only written inside the operation and the last value written to it is kept at the end of the operation), or "*in out*" (the actual parameter is both read and written in the scope of the operation and the last value is kept at the end of the operation; or, in the variant known as "*value-return*", the actual parameter is read in the scope of the operation and updated on return from it).

An example of an operation can be:

operationName(*in* Integer p, *out* Float z)

Later in the design process, interfaces are used to type the provided interfaces (PI) and required interfaces (RI) of component types and are reported in the derived component implementations and then instances.

At instance level, the representation of operations in PI is decorated with attributes that declare the intended concurrent semantics. In our case of interest, we will elaborate on operations with *out* or *in out* parameters which are declared as *sporadic* operations.

A sporadic operation is executed by a dedicated thread of control and there is a guaranteed separation between two subsequent executions of it, called minimum inter-arrival time (MIAT).

Suppose then that we want to target RCM from the CHESS component model and thus generate RCM-compliant entities for the schedulability analysis model and then at implementation.

In RCM an operation with exclusively *in* parameters would be rendered as a composite structure comprised of a thread and a protected buffer. Clients calling the operation would post a request to the designated interface. That request would be reified into an object and posted in the protected buffer. The thread would then be enqueued on a private entry to that buffer. When the buffer is empty, the thread would block. On a non-empty buffer, the thread would fetch the execution request at the logical top of the buffer, according to the queuing policy selected for the buffer, then unfold the invocation object and execute the requested operation. If the thread was marked sporadic, the buffer status becoming not-empty would represent the single source of activation events for the thread and the thread structure would take measures to enforce the MIAT specified for the sporadic operation.

As it is evident, the RCM has a specific and arguably straightforward way of realizing the sporadic semantics of the operation. Another computational model, like that underlying a cyclic executive, would need a radically different way of expressing the event-based sporadic semantics; in the above example, the differences would be needed to reconcile it with the static nature of the time-triggered schedule.

RCM allows us to map an operation with exclusively *in* parameters and sporadic nature on a "sporadic task". In CHESS the actual task implementation would reside in the container-connector level, underneath the PSM.

If the designer specified an operation with *out* or *in out* parameters and wanted to assign a sporadic nature to it, that specification could not be mapped directly to RCM.

That specification would in fact imply that the thread of control of the caller should block, waiting for the completion of the sporadic operation as performed by the thread on the callee side. At tasking level, this would imply a rendezvous (synchronization) between the thread of the caller and the thread of the callee. The fact is, however, that task synchronization cannot be treated by schedulability analysis and it is thus forbidden in RCM. The only way to realize *out* or *in out* semantics would be to create an assembly of RCM-compliant entities that engage in a collaborative call-back pattern that is proven to be analyzable. The definition and implementation of patterns of this kind falls within the charter of WP5 in strict collaboration with WP4, following on from an in-depth analysis of the user requirements and the run-time semantics allowed by the computational models of interest to the industrial users in CHESS. Such patterns would be used first in the PIM to PSM transformations, in order that the implementation model considered for static analysis captures the user requirements and also complies by construction with the restrictions and assumptions stipulated for the analysis. The code level correspondents of those patterns would then be used in the PSM to code transformations in a manner that not only preserves the required semantic compliance but also actively enforces it at run time, if and where necessary. When RCM is chosen as the computational model for the PSM, the component model which underpins the PIM shall be informed of all the constraints in place to ensure the consistency of the target model and the design environment shall enforce them, possibly on the fly.

1.2.2 List of consistency requirements for the Ravenscar Computational Model

Rule	Application level	Kind of application
A sporadic operation is mapped to a thread of control and requires a request buffer protected from mutual exclusion with the immediate ceiling protocol. A blocking operation of the buffer is the single point of suspension for the task. Client of the sporadic operation are simply posting their sporadic request in the protected buffer.	PIM=>SAM and PSM=> source code	Static transformation
A protected operation is mapped to a shared resource protected with the immediate ceiling protocol.	PIM=> SAM and PSM=> source code	Static transformation

Interactions with a sporadic operation as destination imply an access to the protected resource of the sporadic task	PIM => SAM and PSM => source code	Static transformation
An operation with at least one parameter cannot have a cyclic activation pattern	Editor	On-the-fly
An operation with at least one *out* or *in out* parameter cannot have a sporadic activation pattern	Editor	On-the-fly

2. GENERAL SOLUTIONS TO ENSURE CONSISTENCY BETWEEN MODELS AND EXECUTION PLATFORMS

2.1 MODEL SPECIFICATION OF MAPPINGS FROM ANALYSIS PATTERNS TO RUN-TIME PATTERNS

This approach specifies some mapping from both target languages to maintain consistent both transformations, and the analysis and run-time models.

2.1.1 Model specification of mappings from schedulability analysis to violation events

The platform must be suitable for static schedulability analysis. To meet this goal, the adopted computational model should follow the Ravenscar profile. The Ravenscar profile is an industry standard that establishes a set of restrictions to the Ada concurrency model, but can be implemented in other concurrent languages and real-time kernels. The most important restrictions and assumptions include:

- A single processor.
- A statically defined number of threads.
- A single activation event for each thread. The activation event may be generated by the passing of time (for time-triggered threads) or by a signal from either another thread or the environment (for sporadic threads).
- Thread interaction only by means of shared data protected with mutual exclusion locks.

This set of restrictions makes it possible to build systems that include the following kind of PSM components:

- Periodic threads.
- Sporadic threads activated by software level events.
- Sporadic threads activated by hardware level events.
- Protected objects implementing shared data.

- Protected objects used for delivering the activation event to sporadic threads (with just one entry private to the sporadic thread).

These components have been proved in previous projects and industrial development to be expressive enough for implementing high integrity systems for critical applications on a single processor.

In order to preserve the schedulability of the software – if that was asserted by prior analysis, the execution platform must check at run time:

- Deadlines: Each thread must complete its work following a single activation before a given time, i.e., its deadline. The system must be able to detect when a thread overruns its deadline. This control can be achieved, for instance, by means of real-time (interval) timers (watchdogs). If a deadline miss occurs, an event must be raised and some system-level manager must perform some activity for the investigation of the root causes, followed by corrective actions if feasible.

- Worst Case Execution Time (WCET): all threads that have passed static schedulability analysis have declared a worst-case duration for their longest possible activation. The WCET value can be determined by either static timing analysis on the thread's code or by measurement-based observation of actual executions on the designated target. The WCET value is the time the thread would take to complete its longest activation without suffering any interference from the outside. The WCET value consequently represents a bound on the execution time of the thread. The platform must therefore be able to detect when the run-time execution of a thread exceeds its WCET bound. If a violation occurs, the platform must raise an event and some system-level manager shall contain the effects of the timing fault (for example, by preventing further execution of the offending thread).

- The platform must provide mechanisms to detect, at run time, user-defined storage pool overflows. In this case, an exception must be raised.

2.1.2 Model Assumptions of Deployment Determination Analysis

The deployment determination analysis consists of three different steps:

1. Mapping configuration

2. Determination of scheduling priorities

3. Configuration of FlexRay bus systems

The analysis is based on SystemC properties and semantics. In SystemC SC_MODULs can be hierarchically nested. Modules can have ports and attributes. Ports can provide and require interfaces. The implementation of functionality can be realized by SC_METHODs or SC_THREADs. The interaction patters are SystemC blocking and non-blocking calls, events and channels (equivalent to connectors) for outside modules.

The determination of mapping configurations aims to achieve not an optimal but suitable initial mapping configuration. The constraint of interest for this analysis are the maximum resource usage for computational and the communication infrastructure. To

handle complex multi-node systems the analysis does not determine exact bounds for the constraints but tries to estimate good results that are as close as possible to the reality. Therefore there are some points of interest for the execution environment.

The execution platform can validate the determined resource usage bounds in term of utilization of computational hardware and communication infrastructure.

For the analysis, all processes must be statically known. The analysis extracts information from given implementations or from the CHESS model and needs to know the number of instances to estimate the resource utilization.

The button-up approach requires the predefined specification of input data in order to enable the profiling approach that enhances the results of the analysis.

The analysis assumes that every process is activated periodically or sporadically. For sporadic activation the minimum inter-arrival time must be specified.

It is forbidden to implement recursive function calls hierarchies because this is not analyzable by the bottom-up approach at the moment.

It is planned to use the described schedulability analysis for the validation of the determined priorities. Therefore Section 2.1.1 is the reference for this part of the analysis.

The execution environment can check if the calculated bus transfer latencies from the third step of the analysis are held by the system.

2.1.3 Assumptions about simulation analysis to run-time patterns

To perform the timing analysis based on a simulation approach, certain assumptions about the underlying platform are made. The simulation framework is built around the AUTOSAR model which implies certain restrictions. These restrictions must be obvious valid, even if the analyzed model depends on an AUTOSAR environment, which is executed on top of the Linux operating system.

Compared with other analysis methods the simulation-based analysis requires much less rigor. This allows a timing analysis during early stages of the development process and a continuous refinement along the development process. For example, in an early development stage the execution time of a component can be roughly estimated by an expert, further refined and finally estimated on an instruction set simulator before deployed to physical hardware. The following restrictions apply to the simulation-based analysis approach:

- Simulated ECUs only contain a single processor.
- Threads are statically allocated and initialized at system start up.
- Scheduling is restricted to fixed priority pre-emptive and EDF. The latter is a non standard AUTOSAR extension.
- Threads are periodic or sporadic.

- Communication is restricted to automotive networks like CAN, FlexRay and MOST or a generic communication infrastructure.

- No shared data and no synchronization mechanisms between threads on a single ECU.

The simulation framework is build around the AUTOSAR specification making it well suited for the automotive domain as AUTOSAR is there the predominant environment.

2.1.3.1 Restrictions for Thread Management

The target platform is AUTOSAR which runs on top of a Linux system. The AUTOSAR services (threads, etc) are mapped to Linux equivalents. The AUTOSAR environment is under observation of the Linux system to maintain the following assumptions:

- Static thread creation and initialization phase only during application start up.

- The threads must not terminate.

- The threads have a static priority, which remains unchanged.

- Scheduling follows a FIFO-within-priority or round robin policy.

In general, the restrictions and assumptions defined by AUTOSAR and OSEK/VDX apply to the execution model.

2.1.3.2 Assumptions regarding hardware

Regarding the hardware some assumptions are made during simulation. The obvious one is related to the execution time of software components. If the user models an execution time, it depends on a certain underlying hardware which executes the program code. It is not possible to provide some upper bounds on single instructions and catch violations of these during runtime. These violations can only be detected due to exceeded deadline specified by the user (see: User defined assumptions).

The same also holds for network communications. There are assumptions about network latency which occur in CAN, FlexRay or MOST networks depending on the arbitration mechanisms and priorities. Typically it is not possible to validate these assumptions during run-time because the network internals are not exposed to the host due to communication transparency.

As a conclusion it can be said that assumptions about hardware can’t be validated directly during runtime. However violated hardware assumption can appear as a violation of user defined assumptions.

2.2 DEFINITION OF VIOLATION EVENTS TO PRESERVE RUN-TIME CONSISTENCY WITH THE ANALYSIS

2.2.1 Scheduling Analysis: Thread management, synchronization and communication

Threads are an essential component of a real time platform. Platform thread’s management must be simple, predictable and efficient. These characteristics permits a statically analyzable implementation of the platform, both in temporal as in resources

consumption. Furthermore, embedded real time restrictions must be considered, e.g. little memory availability as well as limited CPU performance. Nevertheless, the platform must comply with the Ravenscar profile restrictions:

- All threads must have be created at system initialization. The dynamic creation of threads should be prohibited and disabled by the system. This restriction matches the assumption of classical schedulability analysis; otherwise the analysis becomes too pessimistic to be useful.

- Threads must not terminate. This is symmetric with the restriction that threads are statically created. Threads may and do suspend during their lifetime, but they are not allowed to end. This is because schedulability analysis must be performed against the worst-case contention from a known thread set: this would be difficult to determine if threads could come and go out of existence at will.

- Hierarchical (nested) threads are forbidden because their presence and the time overheads of their creation and activation complicate schedulability analysis very much.

- Communication in a Ravenscar system is restricted to data-oriented communication only. Entry on threads is disallowed; a single entry on each protected object is allowed, provided that only a single thread can enqueue on it. This restriction improves the temporal determinism of the system, permits to determine the worst-case time at which the enqueued thread may be serviced, and simplifies the kernel implementation as well as its efficiency.

- The system must enable the specification of the maximum amount of stack memory that a thread can use. Dynamic memory allocation is forbidden after system initialization. Only scoped or pool based memory allocation are allowed. The use of the general (unbounded) memory pool for dynamic storage allocation is forbidden because its management is not temporally deterministic (it is an NP-problem) and has very high execution time cost.

- Threads must be scheduled with static priorities with a FIFO-within-priority regime. The priority of a thread must not change at run time except for the bounding of priority inversion situations, by use of the immediate ceiling priority protocol. The use of this protocol complies with the Ravenscar profile restrictions and significantly simplifies schedulability analysis. Additionally, it is efficient and simple to implement.

- Thread synchronization must be done through simple algorithms as, for instance, condition variables. The goal of this requirement is to achieve a simple kernel with the least possible overhead and, at the same time, to not introduce temporally non-deterministic operations.

- The platform must support absolute delays. This provides a bounded-drift mechanism for threads to suspend. This requirement is crucial to enable the detection and treatment of WCET and deadlines overruns. Also, absolute delays are essential to build periodic and sporadic activities (i.e. with a minimum inter-arrival time between the releases of subsequent activities).

- The system must support compile-time detection of potentially blocking operations. This feature considerably increases the temporal determinism of the software and simplifies temporal analysis.

2.2.2 Scheduling Analysis: Time management

Time management is essential in real-time systems. The platform must therefore support at least:

- Time-zone independent, monotonic, absolute clock.
- Temporal granularity as high as possible, so that timers can be accurate and fine-grained as needed. High-precision timers allow the schedulability analysis to be much more accurate. The logical tick of the Operating System should therefore be as close as possible to the period of the physical clock, without this adversely affecting the performance of the system.
- The overhead due to interrupt management should be as low as possible.

2.2.3 Scheduling Analysis: Interrupt management

Whenever a peripheral needs the intervention of the CPU, it raises an interrupt. An embedded system may have many peripherals, so that efficient management of interruptions is a key factor to performance. Moreover, interrupt management should be temporarily bounded and must allow the use of priorities and, also, partial or complete inhibition of interrupts.

2.2.4 Simulation Analysis: Violations of User Model Assumptions

The simulation result provides hints as to whether the user requirements concerning deadlines, activations, etc. can be met by the implementation. The result however depends on the fact that the assumptions taken are all fulfilled both during system generation and at run time. Some assumptions are under the control of the developer while other assumptions are implicit, because of the adoption of AUTOSAR and other are derived from the underlying hardware (like network latency).

The developer is responsible for providing trustworthy WCET bounds for the software functions exercised in the system. To assert the consistency between the analyzed model and the execution platform, the run-time environment must provide means to time execution per thread and to fire an alarm if the WCET bound stipulated for that thread is exceeded. The following mechanisms should be provided by the run-time environment to detect violations of user defined assumptions:

- Execution time measurement of software activities.
- Event activation measurement. Events are periodic, sporadic etc. and are subject to jitter which is expressed as an assumption during simulation. Any violation of this prescription (e.g., a sporadic event occurring too often) should be detected and proactively prevented from occurring.

- Synchronization of events can be modelled, i.e. the occurrence of events within a time window. The synchronization groups events or signals and measures the elapsed time between the occurrence of the first and the last event (or signal).

These measurements are usually supported by an integrated timer hardware exploited by the underlying operating system. However, it should be noted that the timer resolution should be at the same magnitude as the user specifies his expression in. The timing mechanism used during simulation allows an almost arbitrary timer resolution (single cycle) which is usually not available during run-time.

2.2.5 Memory management

- The system must have bounded memory usage. To this end, all the memory to be used must be allocated in the system initialization. Dynamic allocation from unbounded storage pool in execution time is forbidden. This is because the dynamic memory management algorithm is temporarily not deterministic. Therefore, it is not possible to perform a schedulability analysis on applications that use this type of memory.

- The system must support volatile memory access, i.e., a method to specify that the variable in question may suddenly change in value.

- The platform must support atomic memory access. In others words, it must be able to specify that the code generated must read and write the type or variable from memory atomically, i.e. as a single/non-interruptible operation. This requirement is essential to build communication and synchronization's thread protocols. An error must be raised if that the platform can't guarantee an atomic access to a variable.

- Virtual memory should be avoided. Virtual memory algorithms are NP complete problems, so bounding response time of memory accesses it not guaranteed.

3. MULTIPLATFORM CONSISTENCY SPECIFICATIONS

This section includes general properties y structure of analysis models generators: i) General structures of generators, ii) integration in general modelling languages and transformation languages, iii) round-trip process and integration of results.

3.1 SCHEDULING

In this section we consider three kinds of generators:

1. Scheduling analysis generator: the input of these generators is UML+MARTE models and the results are MAST models.

2. RTSJ generators: the input of these generators are UML+MARTE models and the result are Java structures (classes and packages) that reuse the RTSJ library and executable in Jamaica VM.

3. Ada 2005 generators: the input of these generators are UML+MARTE models and the results are Ada 2005 structures (tasks, packages and objects), and the Ravenscar profile in particular.

Results for scheduling analysis must be applicable to executable programs generated with generators introduced in 2 and 3. Ada 2005 Ravenscar and RTSJ are designed for the same purposes, and both specifications are designed to make scheduling analysis before programs execution to ensure response times. But each platform has particular properties. Examples include:

- *Memory management.* Ada 2005 Ravenscar excludes dynamic memory allocation. The equivalent approach in RTSJ is allocation of objects in immortal memory, but this kind of restriction would make impossible to reuse multiple library and design patterns. Alternative solutions are scoped memories, but they must be used making consistent scheduling analysis.
- *Time exceptions.* RTSJ supports specific approaches for handling deadline and worst-case execution time exceptions. Ravenscar Ada places important limitations on execution handlers, and the MARTE profile does not include any explicit notation for the description of this kind of handlers, so that the UML modelling concepts must be used instead.
- *Specific design patterns.* RTSJ includes specific patterns such as *asynchronous event handlers* that do not have a direct equivalent in UML+MARTE or Ada Ravenscar. These kinds of patterns will not be considered in generators 2 and 3.

Response time of code generated in generator 2 and executed with Jamaica VM must be consistent with results of MAST scheduling analysis. And the same kind of consistency should be applicable for Ada 2005 and MAST. All three generators must be consistent, and they must be designed and developed taking into account the design patterns of target models/code generated in the other two. UML+MARTE semantics must be the same for all three generators, the code generated in 2 and 3 generators must be consistent with scheduling analysis generator.

4. PLATFORM SPECIFIC CONSISTENCY SPECIFICATIONS

This section introduces the analysis based on code generated. This sections includes details about how to reuse the analysis results in source models, and how to check consistency of models and analysis results

4.1 ADA RAVENSCAR PROFILE/LWCCM PLATFORM

4.1.1 Task management, synchronization and communication

Ada 2005 has direct support for Ravenscar profile by means of a dedicated pragma Profile. When using this compiler directive, the compiler ensures, among others:

- All tasks are statically declared and, consequently, are known at compilation time.
- A Program_Error exception is raised if any task terminates. It also rejects code with abort statement.
- All tasks are declared at library level, so, a hierarchy of tasks is impossible. A compilation error is raised otherwise.
- Entries and accepts can only have one task queued. Otherwise a Program_Error exception is raised. Requeue statement is forbidden as well as select statement since task's queues are not allowed.

- Entries barriers must be simple Boolean expression. This simplifies the evaluation of entries and improves the efficiency of the program.

- Synchronization and communication between tasks are performed through protected objects, which are a high level, safe an efficient mechanism to provide mutual exclusion access to data.

- Pragma Detect_Blocking is used, which provides detection of potentially blocking operations at compilation time.

- System scheduler is established as First Input First Output within priorities

- Only absolute delays (delay until statement) are used.

Moreover, Ada supports:

- pragma Storage_Size that allows defining the maximum amount of stack memory that a task can use. Moreover, Ada support definition of size fixed user defined storage pools in which is allowed to use dynamic memory.

- pragma Priority in the task specifications which permits to establish the priority of the task at compilation time. Ada also provides pragma Locking_Policy. With this pragma is possible to select which policy will be used in the interactions between priority task scheduling and protected object ceilings.

4.1.2 Time management

ORK+ provides a wide support for time management:

- In ORK+, time is represented internally as a 64-bit integer number of ticks. Therefore, the interval of time values that can be represented in this way is approximately -23360..+23360 years.

- LEON2 provides two integer 24 bits timers. With one of them, ORK+ provides the basis for a high-resolution clock used to provide a time zone independent, monotonically increasing, real time clock (called "Real time clock"). The other timer is used to provide a high-resolution timer (General Purpose Timer, in ORK+) which is used to provide the required support for precise alarm handling.

- In order to provide a high resolution clock, the least significant part of the clock is held in the `Real Time Clock hardware register', and the Real Time Clock is programmed to interrupt periodically, updating the most significant part of the clock. As a result, the clock tick is equal to the period of the input signal of the downcounter divided by the prescaler. That is the processor clock period divided by 4 in the current implementation. The clock is a count of ticks and is not synchronized with external sources. As a result, the clock does not jump.

- The overhead due to the interruptions management must be as low as possible and bounded. In ORK+ for 50 MHz LEON2 this overhead is of 4756 processor cycles.

- Clock function must have bounded execution time. In ORK+ the Clock function takes 522 processor's cycles.

- In one hand, each task has attached its execution time timer. This counter is incremented only when a task is running. In the other hand, one task could have multiple real-time timers associated. However, this feature is not useful in Ravenscar because select statement is forbidden; so, only one real-time timer is attached to a task.

- All timing events (pending delays) are queued in a single queue which is ordered by absolute expiration time. A timing event can't be removed from the queue before the delay expires. This helps to keep simple the implementation of the platform.

4.1.3 Interrupt management

The interrupts management in Ada is provided by means of protected procedures, which ensures mutual exclusion access and introduces a very low overhead in the system performance.

4.1.4 Memory management

- Ada supports definition of size fixed user defined storage pools in which is allowed to use dynamic memory. Ada storage pools can be completely customized by the user to adapt it to his requirements.

- Ada provides volatile variables with pragma Volatile. When this pragma is used, the compiler must suppress any optimizations that would interfere with the correct reading of the volatile variables. For example, two successive readings of the same variable cannot be optimized to just one or reordered.

- In Ada is possible to declare a variable as atomic with the pragma Atomic. Atomic implies volatile access, however, since not all types can be access as atomic, the compiler must reject the code if the atomic access of a concrete type it is not supported. When a variable is declared as atomic, the compiler ensures that:

  - The architecture guarantees atomic memory loads and stores,

  - Reordering or suppressing redundant accesses to the object optimizations are disallowed.

4.2 LINUX/OSE PLATFORM

4.2.1 Linux

4.2.1.1 Task management, synchronization and communication

- For real-time systems Linux provides a FIFO and a round robin scheduler (SCHED_FIFO, SCHED_RR) with static real-time priorities. Threads can only be preempted by threads with a higher priority or additionally due to expired time slice (SCHED_RR). To prevent priority inversion the PREEMPT_RT patch introduces priority inheritance mutexes (rt_mutex).

- In general threads can be created dynamically during run-time. However, it is possible to define limits of how many threads can be created. Analysis assumes the infinite execution threads where there are not errors.

- Besides mutexes Linux support futexes (fast userspace mutual exclusion), which are reducing the amount of expensive system calls because most checks are done in userspace.

4.2.1.2 Time management

- Since Linux kernel 2.6.16 the *hrtimer* based infrastructure is included in the mainline. This subsystem offers high resolution timers which are independent of ticks and based on nanoseconds. The time value is stored as plain nanoseconds on 64 bit CPUs and as a seconds, nanoseconds pair on 32 bit CPUs.

- Internally a list of next timer events is kept which are managed via a rb-tree data structure. This decouples the timer system from a fixed system tick (jiffies).

- The hrtimer system can be used in kernel modules (e.g. real-time drivers) directly and also some user space calls are implemented using hrtimer like nanosleep, POSIX timers or itimer. For other timer based services the ticks are emulated by hrtimer.

4.2.1.3 Interruptions management

- To reduce interrupt latencies the PREEMPT_RT patch introduces threaded interrupt handlers. This allows assigning real-time priorities to interrupt handler which are executed as processes. Because the interrupts are executed in process context, they can also be pre-empted by regular processes with higher priorities.

4.2.1.4 Memory management

- Linux supports the use of virtual memory. To prevent that memory of real-time applications is swapped out, the mlockall() call can be used.

- Resource limits can be enforced, limiting the amount of virtual memory, data segment and stack size.

- Applications can allocate dynamic memory via calling malloc().Due to its implementation, it can’t be used on hard real-time systems. Application specific memory allocators can be implemented which are using statically allocated memory or during startup allocated dynamic memory.

4.2.2 OSE

4.2.2.1 Task management, synchronization and communication

- The core concepts in OSE are processes and signals. Inter process synchronization and communication between processes is done through signals (preferably) that processes send and receive (although use of semaphores is possible and supported, but it is not a recommended approach). So sharing

resources is not a recommended and commonly used method in OSE which leads to better reliability and stability of the system.

- Signals can be used to implement other primitives such as semaphores or monitors.

- The fundamental building block in OSE is Process. An OSE process is actually a thread with some special features. An OSE process can be dynamic or static and of type interrupt, timer-interrupt, prioritized, background or phantom.

- Static processes are configured at compile time and are created at the start of the system. It is not allowed to kill a static process.

- In configuring static processes, it is possible to define parameters such as process name, stack size, priority, block to which the process belongs and process redirection table. The process redirection table causes the signals sent to that process to be forwarded to another processes. Since process redirection complicates analysis, it can be avoided in CHESS transformations.

- Creation of dynamic OSE processes can be selectively not used.

- It is possible to group and assign several processes to a process block. Each block can have its own memory pool.

- It is possible to stop and start a process. The kernel keeps a record of how many times a process has been stopped. A process block can also be stopped. (Only prioritized, background and timer-interrupt processes can be stopped. The start and stop calls have no effect on process types that cannot be stopped). Stopping a timer-interrupt process means that it will no longer be scheduled to run at its specified time interval.

- Execution of a process can also be delayed using Delay system call. Delay is not available for interrupt processes.

- Prioritized processes are implemented as infinite loops.

- Phantom processes contain no code and only a signal redirection table and used as a logical channel when communicating across target boundaries (images of a remote process).

- Response-time critical tasks can be defined as interrupt processes.

- Background processes have lowest priority level and used to spend leftover of CPU time

- In waiting for receiving a signal, it is possible to use receive_w_tmo (Receive with timeout) which causes the caller process to be suspended only for the duration specified and continue after expiration of that time.

- It is possible for an interrupt process to see why it was scheduled (using wake_up()). It can be due to hardware interrupt, invoked by a signal or its fast semaphore (each process has one) is signalled.

- A load module in OSE is a file that includes a program's code and data content, and information about how the code and data shall be loaded. This concept can be used as an option in mapping and deployment of components from the CHESS model.

- In OSE, the Main process is responsible for starting all static processes and System Daemon is used for creating and killing processes and blocks. It is possible to change the priorities for these two processes if needed.

- Create handlers are called each time a process is created and there can be several create handlers which in this case will run in undefined order when a process is created. This is important to note for predictability of any mechanism that is built using this feature (this also applies to swap-in handlers).

4.2.2.2 Time management

- Real Time Clock (RTC) is a component[1] in OSE to achieve absolute timing requirements. It is a prioritized process which sets and keeps absolute date and time, generates alarm signals at requested date and time, read the time with the resolution of the operating system, and can be used for converting between different representation of dates and times.

- Time Out Server (TOSV) is a component that allows the user to handle very short duration time intervals. The timer resolution is in milliseconds (although the resolution can be limited to the resolution of the system clock ticks) and up to 50 hours.

- OSTIME (a time given in milliseconds) and OSTICK (Time as reported by get_systime() and get_ticks()) defined types in OSE for timing purposes are declared as unsigned long.

- It is possible to define the time between each system tick using SYSTEM_TIMER configuration parameter. It is target independent and is in milliseconds.

- If an internal timer is available on the CPU chip, it can be used as a system tick timer. The frequency driving the internal timer should be specified. E.g. krn/internal_timer=16500 krn/internal_timer_vector=255

4.2.2.3 Interrupt management

- There are three ways for triggering of an interrupt: hardware interrupt, time interrupt and software event (signal of fast semaphore).

- Among other characteristics (e.g. name) the priority, stacksize, and blockname of an interrupt process can also be defined.

4.2.2.4 Memory management

- Basic type of memory area in OSE is pool. There is always one global memory pool which is the *system pool*. System processes and data reside in this pool.

[1] By term component here we simply mean a part of the OS architecture.

- It is possible to create local pools. Pools can be created dynamically or statically.
- In configuring the system pool, different buffer sizes (e.g. for allocating to signals) and stack sizes are defined.
- A block in OSE may also have its own memory pool.
- It is possible to group one or more memory pools into a *domain* as mechanism for memory protection between processes.
- Memory is allocated and returned after use to a common memory pool.
- In OSE a heap can be created or deleted dynamically.
- Concept of *region*: every accessible logical address in a memory-protected system must belong to a region. A region is either static of dynamic. Accessing an address outside a region leads to an access violation interrupt and eventually an OSE error. It is possible to define R/W/X (execute) permission per regions.
- Free heap buffers are stored in lists according to size. If a buffer of correct size is found in a free, it is removed from the list, initialized and returned to the application requesting memory. If no such buffer is found, the heap locates the closest higher slot with non-empty free-list and splits the first buffer from that list. Maximum time it takes to allocate a buffer from the heap is the time it takes to split buffers all the way from largest size down to the smallest size handled by the heap 23 steps.
- OSE supports three different mappings of memory; i.e. physical vs logical addresses: Single Address Space Equal (SASE, logical addresses are equal to physical), Single Address Space (SAS) and Multiple Address Space (MAS).
- Heap memory in OSE has low overhead of 9/17 bytes (without/with file and line info)

4.3 RTSJ PLATFORM

4.3.1 Task management, synchronization and communication

- The RTSJ allows dynamic thread creation and has no notion of critical time for static thread creation. Therefore, static thread creation has to be emulated. Emulation of static thread creation can be achieved by an initialization thread that gets started from the program's main method, creates and starts all threads of the system, and then terminates. Dynamic thread creation must be disallowed, except in the initialization thread. It would be possible to customize the JamaicaVM for the CHESS project to monitor thread creation and throw a runtime exception if applications attempt to create a thread from outside the initialization thread.
- In the RTSJ, threads are allowed to terminate. To adhere to the requirement that all threads are non-terminating, code generators must only generate non-terminating threads. This can, for instance, be achieved by only generating

threads that are instances of the class *NonTerminatingThread* as sketched in Figure 4.3.1.

```
public abstract class NonTerminatingThread extends RealtimeThread {

   ...  // constructors as in class RealtimeThread

   final public void run() {
     body();
     throw new RuntimeException("Illegal termination of body().");
   }

   /**
    * The actual task body, to be implemented by subclasses.
    * Must not terminate.
    */
   abstract public void body();
}
```

Figure 4.3.1: RTSJ: Emulating non-terminating threads

- In Java, shared resources are protected by object monitors. Conditional synchronization is supported through the primitives *wait()* and *notify()*. Multiple waiters for the same object monitor are allowed. It is, however, not hard to implement a wait-method that throws an exception when called while another thread is already waiting to enter the same monitor. To this end, a counter variable can be used as sketched in Figure 4.3.2.

```
public abstract class ObjectWithAtMostOneWaiter {

   abstract boolean condition();

   private waiters = 0; // Invariant: waiters == 0 || waiters == 1

   public synchronized void waitAtMostOne() {
     if (waiters == 1) {
       throw new RuntimException("Illegal call to waitAtMostOne().");
     }
     waiters++;
     while (!condition()) { wait(); }
     waiters--;
   }
}
```

Figure 4.3.2: RTSJ: Emulating condition synchronization with at most one waiter

- While the RTSJ does not address the configuration of stack sizes, the JamaicaVM allows configuring maximal stack sizes for threads. Java specifies that a *StackOverflowError* has to be thrown when stack size limits are exceeded. The RTSJ makes it possible to avoid garbage-collected heap memory, using immortal memory and scoped memory instead.

- The RTSJ requires a base scheduler that is priority-based. While arbitrary dynamic priority changes are allowed, this feature can be avoided. The RTSJ

specifies the priority ceiling protocol as an optional VM feature. The JamaicaVM supports the priority ceiling protocol.

- For thread synchronization, Java provides the primitives *wait* and *notify*. These can be used for implementing condition variables. Code generators must ensure that synchronization patterns that cannot be analysed are avoided.

- In the RTSJ, absolute delays are supported through the *Timer* class. Timers trigger events at specified times and allow binding asynchronous event handlers to these events. The *Timer* class has two subclasses: *OneShotTimer* and *PeriodicTimer*. A one-shot timer is associated with a single release time, which can be either absolute or relative. In order to delay an action by an absolute time, one can create a one-shot timer with an absolute release time and bind to it the action as an asynchronous event handler. A periodic timer is associated with a single start time, which is either absolute or relative, and with a period, which is relative. An absolute time can also be used to specify the start time of the first release of a periodic thread.

- Java and the RTSJ specify which built-in operations are potentially blocking. The most important ones are calls of synchronized methods, entries to synchronized blocks, calls to *Thread.wait()* and calls to *Thread.join()*. There is no built-in annotation for potentially blocking user-written methods. However, such an annotation could be defined in terms of Java's generic annotation syntax. While Java compilers do not detect potentially blocking methods, it would not be hard to instrument Veriflux to this end, based on whether methods may (transitively) call one of Java's built-in blocking methods.

- The RTSJ does not specify a scheduling order for schedulable objects of equal priority. However, the base scheduler of the JamaicaVM schedules such objects in FIFO order.

4.3.2 Time management

- The RTSJ requires a system real-time clock that is monotonically non-decreasing and measures time with respect to some epoch (e.g., 1 January 1970, 00:00:00, or system start-up time). Time values are represented by a 64-bits millisecond component and a 32-bits nanosecond component. The system real-time clock need not be synchronized with the external world.

- According to the RTSJ, the system real-time clock must progress as uniformly and be as accurate as allowed by the underlying hardware. This implies for instance, that it must not stall and must not be subject to leap ticks.

4.3.3 Interrupt management

- Interrupts from peripherals are not directly addressed by the RTSJ. However, the RTSJ specifies cost monitoring and enforcement, which can be used to deal with rare cost overruns due to interrupts.

4.3.4 Memory management

- The RTSJ permits dynamic memory allocation, including dynamic object allocation on the garbage-collected heap, in scoped memory areas and in

immortal memory, and dynamic creation of new scoped memory areas. In order to establish within an initialization phase a bound on the total memory demand, one can implement an initialization thread that allocates all objects in immortal memory and all scoped memory areas that will be needed throughout the execution, so that further object allocation in immortal memory and further creation of scoped memory areas can be avoid thereafter. Dynamic object allocation within scoped memory areas must still be allowed after the initialization phase, because Java does not permit passing objects or arrays on the call stack. An upper bound on the overall memory demand is determined by the sum of the sizes of the scoped memory areas, plus the size of immortal memory, plus the sum of the stacks sizes for each thread.

- Java allows declaring variables as *volatile*. Thread-shared variables that are accessed without synchronization should always be declared volatile, because otherwise program behaviour is hardly predictable. The Java Language Specification guarantees that a thread *T* observing volatile variables written by another thread *S*, sees their values in an order that is consistent with the order that *S* has written them according to the program text. On multi-processor implementations, the same is not necessarily the case for non-volatile variables.

- The Java Language Specification requires that all variable accesses are atomic, except from accesses to variables of type *long* or *double*, which require two memory accesses. Accesses to *long*s and *double*s should therefore always be synchronized.

- The RTSJ does not require that RTSJ-compliant VMs must disable or avoid virtual memory. The RTSJ offers interfaces for programmers to directly access the kinds of memory that a particular hardware offers.

5. REFERENCES

[1] A. Burns and A. Wellings. HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems. ELSEVIER, 1995.

[2] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. González. A Practitioner's Handbook for Real-Time Analysis: Guide to Monotonic Analysis for Real-Time Systems. Kluwer Academic Publishers, 1993

[3] M. Chen and K. Lin. "A Priority Ceiling Protocol for Multiple-Instance Resource". Proceedings of Real-Time Systems Symposium, 1991.

[4] A. Siebenborn and A. Viehl and O. Bringmann and W. Rosenstiel: "Control-Flow Aware Communication and Conflict Analysis of Parallel Processes", Proceedings of the 12th Asia and South Pacific Design Automation Conference ASP-DAC 2007, Yokohama, Japan 2007

[5] A. Viehl and M. Schwarz and O. Bringmann and W. Rosenstiel: "Performance Risk Analysis at System-Level", CODES+ISSS 2007

[6] A. Viehl and M. Schwarz and O. Bringmann and W. Rosenstiel, "A Hybrid Approach for System-Level Design Evaluation", Embedded System Design: Topics, Techniques and Trends 2007

[7] A. Viehl and M. Pressler and O. Bringmann and W. Rosenstiel: "White Box Performance Analysis Considering Static Non-Preemptive Software Scheduling", Proceedings of the Design, Automation, and Test in Europe Conference (DATE) 2009

[8] M. Joseph and P. K. Pandya, "Finding Response Times in a Real-Time System", The Computer Journal 29(5):390-395, 1986

[9] J.B. Goodenough, and L. Sha, "The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks", Proceedings of the 2nd International Workshop on Real-time Ada issues, pp. 20-31, 1988

[10] K. Tindell, "Adding Time-Offsets to Schedulability Analysis", Technical Report No. 221, Real-Time Systems Group, Department of Computer Science, University of York, York, UK, 1994.

[11] TIMMO Project. (2010, Jan.) TIMMO - Timing Model. [Online]. http://timmo.org/

[12] TIMMO Partners, "TADL: Timing Augmented Description Language version 2", Available http://timmo.org/pdf/D6_TIMMO_TADL_Version_2_v12.pdf, 2009.

[13] N. Fiertag, K. Richter, J. Nordlander, and J. Jonsson, "A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics", in *Proceedings of the IEEE Real-Time System Symposium (RTSS), Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'08)*, Barcelona, 2008.

[14] MSR MEDOC, "Element Attribute Documentation", Available http://msr-wg.de/medoc/download/msrsw/v230/msrsw_v230-eadoc-en/msrsw_v2_3_0.sl-eadoc.pdf, 2005.

[15] The ATESST Consortium, "EAST ADL 2.0 Specification," Available http://www.atesst.org/home/liblocal/docs/EAST-ADL-2.0-Specification_2008-02-29.pdf, 2008.

[16] The Real-Time Specification for Java. http://www.rtsj.org

[17] JSR 282 Expert Group. Realtime Specification for Java, version 1.1, http://jcp.org/en/jsr/detail?id=282. May 2009.

[18] F. Siebert. Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages. aicas Books. 2002.

[19] Jamaica VM 3.4 User Manual. aicas. http://www.aicas.com/documentation

[20] HIJA, High-Integrity Java. Project Number IST-511718 of the 6th framework programme of the European Commission. http://www.hija.info. 2004-2006.

[21] A. Milanova, A. Rountev and B. G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. ACM Transactions on Software Engineering Methodology. 14(1):1-41, 2005.

[22] JSR 308 Expert Group. Annotations on Java Types. Java Specification Request, Java Community Process, http://types.cs.washington.edu/jsr308. November 2009.

[23] M. Tofte, J.-P. Talpin: Region-based Memory Management. Information and Computation(2): 109-176. 1997.

[24] D. Grossman, G. Morissett, T. Jim, M. Hicks, Y. Wang, J, Cheney. Region-Based Memory Management in Cyclone. Proc. of the Conf. on Programming Languages Design and Implementation. 2002.

[25] F. Henglein, H. Makholm, H. Niss. Effect type systems and region-based memory management. In Advanced Topics in Types and Programming Languages, Benjamin Pierce (ed.), MIT Press, 2005.

[26] M. Naik, A. Aiken: Conditional must not aliasing for static race detection. Proc. of the Conf. on Principles of Programming Languages. 2007.

[27] C. Flanagan, K.R.M Leino, M. Lillibridge, G. Nelson, J.B. Saxe, R. Stata. Extended Static Checking for Java. Proc. of the Conf. on Programming Languages Design and Implementation. 2002.

[28] B. Beckert, R. Hahnle, P. Schmitt (eds.). Verification of Object-Oriented Software: The KeY Approach. Springer Verlag, LNCS 4334. 2007.

[29] A. Viehl and M. Pressler and O. Bringmann, “Bottom-Up Performance Analysis Considering Time Slice Based Software Scheduling at System Level”, Embedded Systems Week (CODES-ISSS) 2009

[30] J.C. Palencia, M. González Harbour, "Schedulability Analysis for Tasks with Static and Dynamic Offsets", Proceedings of the 19th IEEE Real-Time Systems Symposium, 1998

[31] J. Gosling, B. Joy, and G. Steele: The Java Language Specification, 3rd Edition. http://java.sun.com/docs/books/jls/

[32] J. Manson, W, Pugh, and S. V. Adve: The Java Memory Model, 32nd ACM Symposium on Principles of Programming Languages, 2005.

[33] OMG CORBA Component Model Specification version 4.0.

[34] A. Burns, B. Dobbing, and T. Vardanega, Guide to the use of the Ada Ravenscar Profile in high integrity systems, University of York, U.K., TR YCS-2003-348, 2003. [Online]. Available: http://www.cs.york.ac.uk/ftpdr/reports/YCS-2003-348.pdf

[35] MAST, Modeling and Analysis Suite for Real-Time Applications, http://mast.unican.es/

[36] AbsInt, aiT Worst-Case Execution Time Analyzers, http://www.absint.com/ait/

[37] D2.1 – CHESS Modelling Language and Editor Version. CHESS Project.

[38] Object Management Group. OMG Unified Modeling Language TM (OMG UML), Superstructure. Version 2.2. http://www.omg.org/spec/UML/2.2/Superstructure

[39] Object Management Group. A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 3. ptc/ 2009-05-13

[40] Object Management Group. UMLTM Profile for Schedulability, Performance, and Time Specification.

[41] A Practitioner's Handbook for Real-Time Analysis. Mark H. Klein, Thomas Ralya, Bill Pollak. Kluwer Academic Publishers. 1993.

[42] EMF: Eclipse Modeling Framework, 2nd Edition. Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks. Addison-Wesley Professional. 2009.

[43] AUTOSAR http://www.autosar.org/

[44] Open SystemC Initiative (OSCI) http://www.systemc.org/

[45] SysXplorer Homepage http://www.fzi.de/index.php/de/component/content/article/238-ispe-sim/4353-sim-tools-sysxplorer

[46] OMG SysML Specification http://www.omgsysml.org/

[47] IP-XACT specification by Spirit Consortium http://www.spiritconsortium.org

[48] Association for Standardizing of Automation and Measuring Systems (ASAM) – FIBEX Specification: http://www.asam.net

[49] A. Viehl, J. Dukadinov, O. Bringmann, W. Rosenstiel: TRANSYSCTOR: A General Methodology and Framework for Rule-Based Transformation and Refactoring of SystemC Designs

[50] André Hergenhan, Wolfgang Rosenstiel: Static Timing Analysis of Embedded Software on Advanced Processor Architectures. DATE 2000

[51] Robert Bosch GmbH, Controller Area Network, http://www.semiconductors.bosch.de/en/20/can/index.asp

[52] Altran GmbH & Co. KG, FlexRay, http://www.flexray.com/

[53] MOST Cooperation, http://www.mostcooperation.com/

[54] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. Technical Report YCS-2003-348, University of York, 2003. http://www.cs.york.ac.uk/ftpdir/reports/YCS-2003-348.pdf.

[55] M. Bordin and T. Vardanega, Correctness by Construction for High-Integrity Real-Time Systems: A Metamodel-Driven Approach, Proc. 12th Int. Conference on Reliable Software Technologies - Ada-Europe, pp. 114-127 , 2007

APPENDIX A. REQUIREMENTS ON PLATFORMS FOR THE CONSISTENCY OF ANALYSIS METHODS

ID (max=80)	Requirement	Platform	Topic	Analysis Method
	General requirements for of analysis on platforms			
1	A cyclic operation is mapped to a thread of control			
2	A sporadic operation is mapped to a thread of control and requires a request buffer protected from mutual exclusion with the immediate ceiling protocol. A blocking operation of the buffer is the single point of suspension for the task. Client of the sporadic operation are simply posting their sporadic request in the protected buffer.	General	Sporadic Events	Scheduling
3	A protected operation is mapped to a shared resource protected with the immediate ceiling protocol.	General	Protected Resource	Scheduling
4	Interactions with a sporadic operation as destination imply an access to the protected resource of the sporadic task	General	Sporadic Events	Scheduling
5	An operation with at least one parameter cannot have a cyclic activation pattern	General	Periodic Events	Scheduling
6	An operation with at least one out or in out parameter cannot have a sporadic activation pattern	General	Sporadic Events	Scheduling
7	Static creation of threads	General	Threads	Scheduling
8	Threads must not finish	General	Threads	Scheduling
9	A single invocation event for each thread. The invocation event may be generated by the passing of time (for time-triggered threads) or by a signal from either another thread or the environment (for sporadic threads).	General	Threads	Scheduling
10	Thread interaction only by means of shared data with mutually exclusive access.	General	Threads	Scheduling
11	Deadlines Checking: Each thread must have its work done before of a given time, i.e., its deadline. The system must be able to detect when a thread is overrunning its deadline. This can be achieved, for instance, by means of real time timers. If this occurs, an exception must be raised and the thread execution must be stopped	General	Platform Check	Scheduling
12	Worst Case Execution Time (WCET) Checking: As schedulable software, all function executions must be bounded and, hence, must have and WCET. The platform must be able to detect when a thread is overran.	General	Platform Check	Scheduling

13	Platform must provide a mechanism to detect, in run time, user defined storage pool overflows. In this case, an exception must be raised.	General	Platform Check	Scheduling
14	Simulated ECUs only contain a single processor.	General	Computing Resource	Simulation
15	Threads are statically allocated and initialized during startup.	General	Threads	Simulation
16	Scheduling is restricted to fixed priority preemptive and EDF. The later is a non standard AUTOSAR extension.	General	Threads	Simulation
17	Threads are periodic or sporadic.	General	Threads	Simulation
18	Communication is restricted to automotive networks like CAN, FlexRay and MOST or a generic communication infrastructure.	General	Threads	Simulation
19	No shared data and no synchronization mechanisms between threads on a single ECU.	General	Threads	Simulation
20	The threads must not finish.	AUTOSAR Linux	Threads	Simulation
21	The threads have a static priority, which remain unchanged	AUTOSAR Linux	Threads	Simulation
22	Scheduling in FIFO or round robin manner	AUTOSAR Linux	Threads	Simulation
23	Hierarchical (nested) threads are forbidden because it complicates schedulability static analyses	General	Threads	Scheduling
24	Thread queues are prohibited; therefore, having more than one thread waiting on a shared resource or for another thread is not allowed.	General	Threads	Scheduling
25	The system must allow defining the maximum amount of stack memory that a thread can use	General	Threads	Scheduling
26	Dynamic memory allocation is forbidden	Ada Ravenscar Profile	Threads	Scheduling
27	Only scoped, inmortal or pool based memory allocation are allowed. The use of the general (unbounded) memory pool for dynamic storage allocation is forbidden	Java-RTSJ	Threads	Scheduling
28	Threads must support static priorities. The priority of a thread must not change except to avoid priority inversion, in which case, only the use of immediate ceiling priority protocol is allowed	General	Threads	Scheduling
29	Thread synchronization must be done through simple algorithms as, for instance, condition variables	General	Threads	Scheduling
30	The platform must support absolute delays. This provides a mechanism to suspend a thread leaving the processor free for others threads	General	Threads	Scheduling
31	The system must support potentially blocking operations detection at compilation time	General	Threads	Scheduling
32	The system scheduler must be First Input First Output within priorities	General	Threads	Scheduling

33	Time zone independent, monotonically increasing, absolute clock	General	Time	Scheduling
34	The temporal granularity of the system should be as high as possible, so that timers can be established as accurately as possible	General	Time	Scheduling
35	The overhead due to the interruptions management must be as low as possible	General	Time	Scheduling
36	Interrupt's management should be temporarily bounded and must allow the use of priorities and, also, partial or complete inhibition of interruptions	General	Interruption Handlers	Scheduling
37	Execution time measurement of modeled components	General	Platform Check	Simulation
38	Event activation measurement. Events are periodic, sporadic etc. and are subject to jitter which is expressed as an assumption during simulation	General	Platform Check	Simulation
39	Synchronization of events can be modeled, i.e. the occurrence of events within a time window. The synchronization groups events or signals and measures the elapsed time between the occurrence of the first and the last event (or signal).	General	Platform Check	Simulation
40	The system must have bounded memory usage. To this end, all the memory to be used must be allocated in the system initialization	General	Memory	Scheduling
41	System must support volatile memory access, i.e., a method to specify that the variable in question may suddenly change in value	General	Memory	Scheduling
42	The platform must support atomic memory access. In others words, it must be able to specify that the code generated must read and write the type or variable from memory atomically, i.e. as a single/non-interruptible operation	General	Memory	Scheduling
43	Virtual memory mechanism should be avoided			
	Ada-2005 Ravenscarn platform			
44	All tasks are statically declared and, consequently, are known at compilation time	Ada Ravenscar Profile	Tasks	Scheduling
45	A *Program_Error* exception is raised if any task finishes	Ada Ravenscar Profile	Tasks	Scheduling
46	All tasks are declared at library level, so, a hierarchy of tasks is impossible	Ada Ravenscar Profile	Tasks	Scheduling
47	Entries and accepts can only have one task queued	Ada Ravenscar Profile	Tasks	Scheduling

48	Entries barriers must be simple Boolean expression	Ada Ravenscar Profile	Tasks	Scheduling
49	Synchronization and communication between tasks are performed through protected objects, which are a high level, safe an efficient mechanism to provide mutual exclusion access to data	Ada Ravenscar Profile	Tasks	Scheduling
50	Pragma *Detect_Blocking* is used, which provides detection of potentially blocking operations at compilation time	Ada Ravenscar Profile	Tasks	Scheduling
51	System scheduler is established as First Input First Output within priorities	Ada Ravenscar Profile	Tasks	Scheduling
52	Only absolute delays (delay until statement) are used	Ada Ravenscar Profile	Tasks	Scheduling
53	Ada provides pragma Storage_Size that allows defining the maximum amount of stack memory that a task can use	Ada Ravenscar Profile	Tasks	Scheduling
54	Ada provides pragma Priority in the task specifications which permits to establish the priority of the task at compilation time	Ada Ravenscar Profile	Tasks	Scheduling
55	In ORK+, time is represented internally as a 64-bit integer number of ticks. Therefore, the interval of time values that can be represented in this way is approximately -23360..+23360 years	Ada Ravenscar Profile	Time	Scheduling
56	LEON2 provides two integer 24 bits timers. With one of them, ORK+ provides the basis for a high resolution clock used to provide a time zone independent, monotonically increasing, real time clock (called “Real time clock”)	Ada Ravenscar Profile	Time	Scheduling
57	In order to provide a high resolution clock, the least significant part of the clock is held in the `Real Time Clock hardware register', and the Real Time Clock is programmed to interrupt periodically, updating the most significant part of the clock	Ada Ravenscar Profile	Time	Scheduling
58	The overhead due to the interruptions management must be as low as possible and bounded. In ORK+ for 50 MHz LEON2 this overhead is of 4756 processor cycles	Ada Ravenscar Profile	Time	Scheduling
59	Clock function must have bounded execution time. In ORK+ the Clock function takes 522 processor’s cycles	Ada Ravenscar Profile	Time	Scheduling
60	In one hand, each task has attached its execution time timer. This counter is incremented only when a task is running	Ada Ravenscar Profile	Time	Scheduling

61	All timing events (pending delays) are queued in a single queue which is ordered by absolute expiration time.	Ada Ravenscar Profile	Time	Scheduling
62	Ada support definition of size fixed user defined storage pools in which is allowed to use dynamic memory	Ada Ravenscar Profile	Memory	Scheduling
63	Ada provides volatile variables with pragma Volatile. When this pragma is used, the compiler must suppress any optimizations that would interfere with the correct reading of the volatile variables	Ada Ravenscar Profile	Memory	Scheduling
64	In Ada is possible to declare a variable as atomic with the pragma Atomic	Ada Ravenscar Profile	Memory	Scheduling
	Java-RTSJ platform			
65	The RTSJ allows dynamic thread creation and has no notion of critical time for static thread creation. Therefore, static thread creation has to be emulated	Java-RTSJ	Threads	Scheduling
66	In the RTSJ, threads are allowed to terminate. To adhere to the requirement that all threads are non-terminating	Java-RTSJ	Threads	Scheduling
67	In Java, shared resources are protected by object monitors	Java-RTSJ	Threads	Scheduling
68	While the RTSJ does not address the configuration of stack sizes, the JamaicaVM allows to configure maximal stack sizes for threads	Java-RTSJ	Threads	Scheduling
69	The RTSJ requires a base scheduler that is priority-based. While arbitrary dynamic priority changes are allowed, this feature can be avoided.	Java-RTSJ	Threads	Scheduling
70	For thread synchronization, Java provides the primitives wait and notify. These can be used for implementing condition variables	Java-RTSJ	Threads	Scheduling
71	In the RTSJ, absolute delays are supported through the Timer class	Java-RTSJ	Time	Scheduling
72	Java and the RTSJ specify which built-in operations are potentially blocking	Java-RTSJ	Threads	Scheduling
73	The RTSJ does not specify a scheduling order for schedulable objects of equal priority. However, the base scheduler of the JamaicaVM schedules such objects in FIFO order	Java-RTSJ	Time	Scheduling
74	The RTSJ requires a system real-time clock that is monotonically non-decreasing and measures time with respect to some epoch	Java-RTSJ	Time	Scheduling
75	According to the RTSJ, the system real-time clock must progress as uniformly and be as accurate as allowed by the underlying hardware	Java-RTSJ	Time	Scheduling
76	Interrupts from peripherals are not directly addressed by the RTSJ. However, the RTSJ specifies cost monitoring and enforcement, which can be used to deal with rare cost overruns due to interrupts	Java-RTSJ	Threads	Scheduling

77	The RTSJ permits dynamic memory allocation, including dynamic object allocation on the garbage-collected heap and in immortal memory, and dynamic creation of new scoped memory areas	Java-RTSJ	Memory	Scheduling
78	Java allows to declare variables as volatile	Java-RTSJ	Memory	Scheduling
79	The Java Language Specification requires that all variable accesses are atomic, except from accesses to variables of type long or double, which require two memory accesses	Java-RTSJ	Memory	Scheduling
80	The RTSJ does not require that RTSJ-compliant VMs must disable or avoid virtual memory. The RTSJ offers interfaces for programmers to directly access the kinds of memory that a particular hardware offers	Java-RTSJ	Memory	Scheduling