Validating requirements at model-level

Olivier GILLES[1], **Jérôme HUGUES**

GET-Télécom Paris – LTCI-UMR 5141 CNRS
46, rue Barrault, F-75634 Paris CEDEX 13, France
{olivier.gilles,jerome.hugues}@enst.fr

ABSTRACT. *Designing an embedded real-time system is a complex process, which involves modeling, verification, validation of system requirements. In this paper, we present how to integrate a Domain Specific Language to the AADL — REAL — to perform requirements definition and their validation of complete systems at model level. We also present how to use REAL to check models in an efficient manner, and give some examples of requirements that should be enforced. We conclude by presenting the process in which it can be used with a code generator in order to get requirement-compliant source code for the targeted application.*

RÉSUMÉ. *Concevoir un système embarqué temps-réel est un processus complexe, qui implique des phases de modélisation, de vérification, et de validation des besoins d'une application. Dans cet article nous présentons comment intégrer un "Domain Specific Language" à AADL, pour définir des contraintes et les valider sur des systèmes complets, au niveau du modèle. Nous présentons également comment utiliser REAL pour vérifier les modèles de manière efficace, et donnons quelques exemples de contraintes qui devraient être vérifiées dans un modèle temps-réel. Nous concluons en présentant la processus dans lequel il peut être utilisé avec un générateur de code pour obtenir un code source respectant les contraintes définies pour l'application cible.*

KEYWORDS: *embedded systems, real-time, critical, verification, model, architectural languages*

MOTS-CLÉS : *systèmes embarqués, temps-réel, systèmes critiques, vérification, modélisation, ADL*

1. This work is funded in part by the ANR Flex-eWare project.

e soumission à *IDM'08*, le June 18, 2008

1. Introduction

Distributed Real-time and Embedded (DRE) systems must enforce constraints of both the real-time and embedded domains. They need to comply with specific requirements: strict determinism, low resource consumption and reliability.

Designing DRE systems is a complex and thus expensive task, which requires highly specialized manpower. Some methodologies have been developed in order to reduce DRE development costs, amongst them, model-driven design (MDD) coupled with code generation appear to be one of the most promising in reducing the development cost and increasing reliability.

DRE systems must enforce a large set of non-functional requirements that come from their versatile nature. A real-time application needs strong timing guarantees; a distributed one must not break those guaranties. An embedded application also needs to check resource usage.

Hence, enforcement of non-functional requirements is required to assess DRE system consistency. Yet, the complexity of expression of these constraints is directly proportional to the level of restrictiveness of the non-functional requirements.

Therefore, one need an efficient way to express these requirements on the model prior to evaluating them. One need also to make sure there is no divergence between the model and the requirements put on this model.

In this paper, we present a solution to these two problems for AADL-centric processes, using a Domain Specific Language. This DSL checks non-functional constraints at model level, ensuring that the model is ready for further analysis (e.g. schedulability) or code generation for constrained run-times.

2. Architecture Description Languages and non-functional requirements

In this section, we present the place of non-functional requirements definition within the software development process and then the works related to non-functional requirement definition and verification.

2.1. *Software development process*

The software development process maps requirements onto high-level, low-level and code-source level descriptions. At each level, functional and non-functional requirements must be refined, faithfully translated to the actual system description, and also preserved. Some of them can be also checked at each stages.

Architecture Description Languages (ADLs) (Medvidovic *et al.,* 1997) aim at describing the system architecture. An ADL-based development process consists of modeling the application, analyzing it and then generating source code. ADL tool-

suites can also provide support for describing system requirements for further validation and verification. Some examples of such approach are STOOD (Dissaux, 2004), OSATE (team, 2004) and TOPCASED (Farail *et al.,* 2005).

DRE systems have complex non-functional requirements. While some of them are generic (and solved using well-known design patterns, e.g. concurrency (Douglass, 2002)), most of them are highly application-dependent. Hence, it is necessary to have a language which can express application requirements. ADLs can document low-level architectures, but not constraints on them. Therefore, defining a language as a Domain Specific extension of an ADL would enable one to describe both software architecture and requirements in the same model. A checker for this language would validate model conformance to these requirements. This would simplify code generation from specific ADL patterns.

2.2. *Related work*

Non-functional requirements definition and enforcement can be performed by adding constraints to the model. UML (OMG, 2003) and its derivatives (MARTE (OMG, 2007)) are the de jure standards. Object Constraint Language (Warmer *et al.,* 1998) (OCL), is a standard to express constraints on UML models. OCL constraints are described over the concepts of a meta-model and evaluated over a model which is an instance of the meta-model. OCL requires a generic API to express queries, leading to complex expressions for assessing details of a model. This indirection to a higher level of abstraction increases the learning curve.

Approaches exist to map a model onto other formalisms for verification purposes, e.g. using algebraic approaches like Z for modeling RT-POSIX (Freitas, 2006); or model checking techniques for verifying RT-CORBA middleware using the Bogor model checker (Deng *et al.,* 2003). Yet, this introduces a new modeling space to master, especially for domain engineers. One need to automate the mapping between these two spaces. Besides, it could introduce inconsistencies in the process when one model is updated, but not the other. This could lead to inadequate or incomplete models. Therefore, one need to add requirements directly on the model to ensure analysis techniques are still applicable, and that the requirements apply to an up-to-date model.

Therefore, we propose a DSL mapped to the concept of meta-model, while being simple enough to reduce risk of inconsistency and learning time. Using mathematical notations from set theory would make this language easier to manipulate. In this paper, we present a solution and its implementation on top of the AADL. AADL already has supporting tool for schedulability analysis, code generation. Complementing it with a DSL for specifying constraints would ensure AADL models are ready for being processed by such tools in a transparent way.

3. Short overview of AADL and OCARINA

AADL (*Architecture Analysis and Description Language*) (SAE, 2004) aims at describing DRE systems by assembling components. AADL allows for the description of both software and hardware parts of a system. It focuses on the definition of interfaces, and separates the implementations from these interfaces.

An AADL description is made of *components*. The AADL standard defines software components (`data`, `thread`, `thread group`, `subprogram`, `process`) and execution platform components (`memory`, `bus`, `processor`, `device`) and hybrid components (`system`). Components describe well identified elements of the actual architecture. *Subprograms* model procedures as in C or Ada. *Threads* model the active part of an application (such as POSIX threads). AADL threads may have multiple operational modes. Each mode may describe a different behaviour and property values for the thread. *Processes* are memory spaces that contain the *threads*. *Processors* model micro-processors and a minimal operating system (mainly a scheduler). *Memories* model hard disks, RAMs, *buses* model all kinds of networks, wires, *devices* model sensors, etc.

An AADL model also describe non-functional facets: embedded or real-time characteristics of the components (execution time, memory footprint...), behavioral descriptions, etc. Description can be extended either through new property sets, or through annexes. Annexes are extensions to the core language. A complete introduction to the AADL can be found in (Feiler *et al.,* 2006).

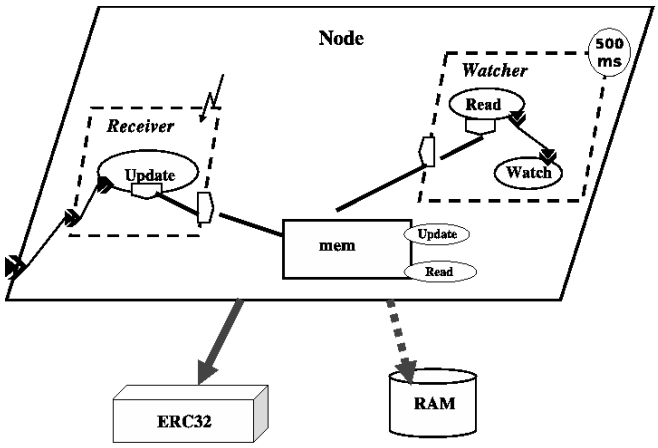

Figure 1. *Sample* AADL *model*

Figure 1 is a sample AADL model. It models two threads: one periodic and one aperiodic that interact to read and update a shared variable. Both threads are subcomponent of a process, bound to a processor and memory.

We have developed the OCARINA (Vergnaud *et al.,* 2006) toolsuite to manipulate AADL models. OCARINA proposes AADL model manipulation based on a compiler-like API. "Back-end" modules can generate formal models, perform scheduling analysis and generate distributed high-integrity applications in Ada.

Generated code relies on the PolyORB-HI (Hugues *et al.,* 2007) middleware to ensure communications and task allocation. PolyORB-HI ensures that a minimal and reliable middleware is generated for a given distributed application.

AADL is a complete ADL that describes in -depth the architecture of a system. It allows for layered design through component refinement. Besides, the designer can express all is non-functional requirements. We now detail how we enriched AADL with a DSL to check non-functional properties at model level.

4. REAL

REAL[1] (Requirement Enforcement Analysis Language) aims at checking adequacy between different parts of architectural descriptions, with emphasis on conciseness and simplicity. In this section, we describe the main features of this language.

4.1. *Basic constructs*

REAL is based on set theory. It allows one to build sets whose elements are AADL entities (connections, components or subprogram calls). Verification can then be performed on either a set or its elements by stating Boolean expressions. The basic unit of REAL is a *theorem*. A theorem is made of 3 parts: range definition, destination set building and the verification expressions. We now review each of them.

4.1.1. *Range definition*

The *range definition* selects the class of component instances on which the verification must be done. All following declarations (either set building or verification expressions) are performed for each element of the range set. An element of the range set is a *range variable*.

```
thread Receiver
features
   mem : requires data access shared_data.i;
properties
  Dispatch_Protocol => Sporadic;
  ARAO::Priority => 2;
  RTOS_Properties::Criticity => 3;
end Receiver;

thread Watcher
features
  mem : requires data access shared_data.i;
```

1. REAL sources and documentation can be accessed from http://aadl.enst.fr/ocarina/real.html

```
properties
  Dispatch_Protocol => Periodic;
  Period            => 500 Ms;
  ARAO::Priority => 4;
  RTOS_Properties::Criticity => 3;
end Watcher;

process node end node;

process implementation node.i
subcomponents
  th1 : thread Receiver;
  th2 : thread Watcher;
  sh_mem : data shared_data.i;
connections
  data access sh_mem -> th1.mem;
  data access sh_mem -> th2.mem;
end node.i;
```

Listing 1: AADL threads

Listing 1 is an excerpt of the AADL model in figure 1. In this example, let us assume that a range set is declared as being formed of the predefined thread set.

```
theorem Thread_Range_Set

foreach e in Thread_Set do
  -- Here is theorem code
end Thread_Range_Set;
```

Listing 2: Thread periodicity

In this case (listing 2), the range variable *e* would be successively valued as the first thread (*Receiver*) in the first evaluation iteration, then to the second one (*Watcher*).

4.1.2. *Destination set building*

Destination set building defines sets that can refer to previously-defined ones. Additionally, they can refer to predefined sets (canonical one, e.g. all processors in a model). Their elements can be declared to be compliant to certain properties. These definitions (or *relations*) are defined this way: `S := x in E | f (x)`, where the character '`|`' is an abbreviation for `such as`. In REAL, it means that the destination set will be formed by all elements of the set `E` which verify the expression `f(x)` (which is an application from the set E to the boolean).

This means that `S` holds all the elements of the set *E* that comply with the relation *f*. Hence, set building is done by giving its first-order logic definition.

```
data shared_data
properties
  Concurrency_Control_Protocol => Protected_Access;
  ARAO::Priority => 5;
end shared_data;

data implementation shared_data.i
end shared_data.i;
```

Listing 3: AADL protected data

Let us suppose we want to build the set of concurrency-safe data (i.e. for which the AADL property *Concurrency_Control_Protocol* is set (listing 3)). The set of concurrency-safe data components is :

```
Protected_Data_Set :=
 {x in Data_Set |
       Compare_Property_Value
         (x, "Concurrency_Control_Protocol", "Protected_Access")}
```

Listing 4: access-protected set building

4.1.3. *Verification expression*

Verification expressions check properties on sets, according to the range variable. If it refers to the range variable, or if it refers to any set that refers to the the range variable, it will be evaluated for each element of the range set.

For listing 1, one might want to check whether all threads are periodic. Using the check in listing 5, the evaluation will fail on the first element *e* (the *Receiver* thread), but would have been successful on the second one (the *Watcher* thread).

```
theorem Only_Periodic_Thread

foreach e in Thread_Set do

  check (Get_Property_Value (e, "Dispatch_Protocol") =
         "periodic");

end Only_Periodic_Thread;
```

Listing 5: Thread periodicity

4.2. *Relations*

As exposed in Section 4.1.2, sets can be defined that refer to previously defined sets. Sets gather elements matching specified properties. REAL defines relations to find hierarchical links between AADL component instances. Relations allow to navigate in the AADL model and thus to access AADL architectural semantics within REAL. For instance, the relation `A is_subcomponent_of B` returns `true` whenever the instance A is a subcomponent of the instance B. Relation can be expressed between two elements, or between a set and an element. In that case, this example would return true whenever A is a subcomponent of any element of the set B.

4.3. *Advanced features*

4.3.1. *Flow analysis*

Aside from the REAL interpreter, we have developed *Flow Analysis Tool* (FAT), a tool which allows to deduce AADL implicit flows from the AADL model, and to store them back in the OCARINA tree. Hence, end-to-end flows can be designated

with REAL, thus allowing to check flow-related constraints in an efficient and concise manner. An example of such constraint is to check the bound of flow latency.

4.3.2. REAL & AADL *annexes*

A REAL theorem can be declared either in a REAL file or in annexes within AADL components.

When a REAL theorem is declared in an AADL component, the component's instances form the *theorem context*, the theorem itself being *contextualized*. Practically, a set named the *local set* that contains all the instances of the component will be generated. It allows the direct addressing of a kind of peculiar component instead of the whole class (eg. check a property only on SPARC processors instead of all processors).

A theorem defined in a REAL file is named *contextfree*. Such theorems can refer to the *local set*, but without having a priori any information on the kind of its elements (cf. Section 4.3.3). The main usage of REAL files is to define libraries for contextualized theorems.

4.3.3. *Required theorems*

REAL allows the definition of other theorems as *required* for proving the current one. Those theorems are evaluated before the current one. If any of those *required theorems* is false, then REAL will tag the current theorem as false.

When a required theorem is evaluated, its context (ie. its *Local Set*) is inherited from the callee theorem. It implies that required theorems must not be defined within an AADL annexe, but in REAL files.

4.4. *Implementation*

We have implemented a REAL parser, checker and verifier within OCARINA which can work with separate REAL files as well as REAL annexes of AADL components. Manipulations are performed on OCARINA's Abstract Syntax Tree (AST). A set being a collection of entities from the AST. The REAL checker works on the internal view of the AADL model to perform validation.

Let us note a REAL theorem puts constraints on the structure of the model. Its verification implies building subsets of the AST impacted by the theorem, and performing set operations on them (union, intersection, iteration, etc.). This gives an upper-bound on the resources used by the operation.

Hence, memory use for proving a REAL theorem is bounded by the complexity of the AADL model and the number of sets actually declared in the theorem. A subset references entities from the AST. Therefore, its size is at most proportional to that of the AST. It usually consumes less memory.

All operations within REAL are either unary or binary. In a worst-case scenario, both operands of an operation are sets (rather than being a set/element or element/element couple), and both sets may have the same size as the number of elements in the original AADL model, in which case the operator will be applied a total of NxN times. Thus, the upper bound for the execution time of the algorithm is quadratic.

5. Using REAL to validate models

Two kind of constraints can appear in a model : runtime constraints and model-specific constraints. The runtime constraints must be enforced in all models fit in the constraint, while model-specific constraints must be enforced regardless of the actual runtime. In this section, we illustrate both by giving an example, and then the related REAL theorem.

5.1. *A runtime-specific set of constraints : the Ravenscar Profile*

In this section we illustrate the use of REAL on a complete example: ensuring that an AADL model complies to the Ravenscar Profile. We first give a quick definition of the Ravenscar Profile, and then we explain how to ensure that the model is compliant to this profile, with the help of REAL and AADL. For each described theorem, we give a short interpretation for the model corresponding to Figure 1.

The Ravenscar Profile (Burns *et al.,* 2003) targets real-time and critical systems. It is a subset of the Ada language that restricts concurrency constructs that prevent schedulability analysis. In particular, strong restrictions are put on communication and runtime constructs such as *tasks, rendezvous* and *protected objects*. Basically, this profile forbids any dynamic and non-deterministic features in concurrent programming. It has also been adapted for RTSJ (Kwon *et al.,* 2005).

5.1.1. *Thread and protected objects restrictions*

To be Ravenscar-compliant, the code must comply to the following properties :

– *No aperiodic tasks* : Threads are either sporadic or periodic, scheduled by the `FIFO_Within_Priorities` policy.

– *PCP-consistent* : concurrent access to shared data uses the Priority Ceiling Protocol (Sha *et al.,* 1990). This protocol ensures mutual exclusion while also bounding priority inversion.

Other restrictions are more specific to the code being used, and cannot be checked at model level (e.g. use of time-related functions).

By defining these properties as REAL rules, one can ensure an AADL model matches the constraints of the target runtime. This ensures earlier detection of model errors, but also reduce the complexity of checks to be performed by the code generator.

Therefore, one can generate code that follows the architectural description with an code generator like OCARINA/PolyORB-HI for High-Integrity systems.

We now detail how to check the Ravenscar properties.

5.1.2. *Absence of aperiodic tasks*

5.1.2.1. AADL translation

The first step toward writing a REAL theorem is to map the code-related statement in the AADL model. The statement is natural : the `Dispatch_Protocol` AADL standard property defined the nature of the thread. Hence, verifying whether tasks are periodic or sporadic is the same as verifying if threads have the property `Dispatch_Protocol` set to `Sporadic` or `Periodic`.

5.1.2.2. REAL translation

In listing 6, `Thread_Set` is a predefined set that includes all AADL thread component instances. `Get_Property_Value` returns the value of the property for the given element. In this case, the value is tested for each element of the *range set*.

```
theorem task_periodicity

foreach t in Thread_Set do

   check ((Get_Property_Value (t, "Dispatch_Protocol") =
                              "periodic") or
          (Get_Property_Value (t, "Dispatch_Protocol") =
                              "sporadic"));

end task_periodicity;
```

Listing 6: Task Periodicity

5.1.2.3. Interpretation

In Listing 1, the threads `Receiver` and `Watcher` are part of the process `node.i`. Since `Receiver` is sporadic and `Watcher` is periodic, the theorem is verified for this model.

5.1.3. *PCP-Compliance*

5.1.3.1. AADL translation

For a model to be compliant with PCP hypothesis, one has to check two conditions:

– Shared data components use the PCP concurrency control mechanism;

– No data component following PCP has an accessor thread whose priority is superior to the ceiling priority and all those threads are hosted by the same processor.

The first condition is equivalent to assessing that if more than one thread access a data component instance (ie. the same data component instance is

provided to those threads), then the data component instance must have the `Concurrency_Control_Protocol` set to `priority_ceiling`.

The second condition states: all threads accessing a data must have a priority (ie. user-defined property `Priority`) which is less than the priority of the accessed data. Furthermore, all threads must be on the same processor.

5.1.3.2. REAL translation

The theorem is split in two parts: the theorem in Listing 7 checks whether PCP has been declared for all shared data; the second one (listing 8) checks whether conditions for PCP are present. We will use some predefined sets and relations:

– `Data_Set` contains all data instances

– `Processor_Set` contains all processor instances

– `Is_Accessing_To` relation returns true when the first argument accesses the second one.

– `Is_Bound_To` relation returns true when the first argument's `Actual_Processor_Binding` property refers to the second argument.

```
theorem all_pcp

foreach d in Data_Set do

   accessor_threads := {t in Thread_Set | Is_Accessing_To (t, d)}

   check (Cardinal (accessor_threads) = 1 or
          (Get_Property_Value
             (d, "Concurrency_Control_Protocol") =
             "Priority_Ceiling"));
end all_pcp;
```

Listing 7: Shared data access

```
theorem PCP

foreach e in Data_Set do

  -- Set of the threads that acceed to the protected data
  accessor_threads := {t in Thread_Set | Is_Accessing_To (t, e)}

  --  set of the processor(s) that the
  --  accessor_threads are bound to
  threads_processors := {p in Processor_Set |
                      Is_Bound_To (accessor_threads, p)}

  --  call PCP theorem
  requires (pcp);

  --  proceed to the actual verification
  check (((Get_Property_Value
             (e, "Concurrency_Control_Protocol") <>
           "Protected_Access") or
     (Get_Property_Value (e, "ARAO::Priority") >=
            Max (accessor_threads,
                 Get_Property_Value,
                 "ARAO::Priority")))
          and Cardinal (threads_processors) <= 1);
```

```
end PCP;
```

Listing 8: PCP

5.1.3.3. Interpretation

Listings 1 and 3 show that the only data which is accessed by more than one thread is `sh_mem` of type `shared_data`, which does have the `Concurency_Control_Protocol` set to the value `Protected_Access`, thus verifying theorem 7. In the AADL model, we also see that the threads `Watcher` and `Receiver` priorities are respectively 2 and 4. Since only those threads access to the shared data, and the latter priority is 5, the theorem 8 is verified too. We can conclude that the AADL model is compliant to PCP.

5.2. *Model-specific constraints*

Model-specific constraints are diverse by nature. In this section, we present two of them : secure communications and task fitness. For both of them, we present a short description, the meaning in the AADL model and the REAL theorem corresponding to the constraint.

5.2.1. *Secure communication*

Secure communication is a condition requested by the standard API ARINC 653 part 1 (ARINC, 1997). It's a commonly used standard in avionic systems to ensure a strict separation of the processes, with different level of separation according to the process criticity. One of the condition that the system must comply to is that a thread must not interfere with a thread of superior criticity.

5.2.1.1. AADL translation

A `ARAO::criticity` property has been defined in AADL. A communication from a thread to another one (ie. a message sent) is modeled by a `connection` between the corresponding component instances.

5.2.1.2. REAL translation

In Listing 9, `Thread_Set` is a predefined set that includes all AADL thread component instances. `Is_Connecting_To` returns all threads which receive data or signals (ie. AADL `connections`) from the current one. `Get_Property_Value` returns the value of the property for the given element. In this case, the value is tested for each element of the `range set`.

```
-- Check a condition of ARINC security :
-- for any thread, all threads which send it messages must have an
-- equal or less criticity

theorem ARINC_Secure
```

```
foreach e in Thread_Set do

   Cnx_threads(e) := {t in Thread_Set | Is_Connecting_To (e, t)}

   check (Get_Property_Value (e, "RTOS_Properties::criticity") >=
          Max (Cnx_threads, Get_Property_Value, "RTOS_Properties::criticity"));

end ARINC_Secure;
```

Listing 9: Task isolation

5.2.1.3. Interpretation

In Listing 1, the threads `Receiver` and `Watcher` are of the same criticity. Yet, they have no direct connections, hence they comply to the constraint.

5.2.2. *Correct use of a subprogram*

A frequent requirement of embedded systems is to follow string period and priority assignment for threads running mathematical functions (e.g. Kalman filters, Runge-Kunta functions to control a systems). We present here how such constraints can be dealt with AADL and REAL.

5.2.2.1. Implementation

Such a constraint on a subprogram is strictly local, and must be designed separately for each subprogram. REAL allows the designer to define a theorem relatively to a given component, here a subprogram. As described in Section 4.3.2, it consists in placing the theorem within the component's `annex`, and then using the `Local_Set` as a reference to it.

We used the REAL relation `Is_Calling`, which returns the components instances that are called by the parameter-passed component instance. The `Period` AADL property defines the period of a thread, in milliseconds.

The Listing 10, complete the model defined by Listing 1 by defining more precisely the `Watcher` thread. It now calls the subprogram `watch`. The REAL theorem written in the subprogram annex checks whether the thread which it is run by has the right period and priority.

```
subprogram watch
features
  data_in : requires data access shared_data.i;
properties
  source_language => Ada95;
annex
  min_req {**
   theorem subprogram_requierements

   foreach t in Thread_Set do

      Called := {l in Local_Set | Is_Calling (t, l)}

      check ((Cardinal (Called) = 0) or
             ((Get_Property_Value (t, "ARAO::Priority") = 4)
               and (Get_Property_Value (t, "Period") = 500)));
```

```
    end subprogram_requierements;
  **};
end watch;

thread implementation Watcher.impl
calls {
  sp : suprogram watch;
};
connections
  data access mem -> sp.data_in;
end Watcher.impl;
```

Listing 10: Subprogram constraints

5.2.2.2. Interpretation

First, the theorem execution build the intermediary set `Called`. It is composed of a subset of the `Local_Set` defined by the component which are called by the current `range element`, a thread component. Called on any subprogram, it will return the subprogram, if it is called by the current `range element`, or else be empty. In our case, `watch` is present in the `Local_Set`, hence the `called` set will be successively containing `watch` (when thread `Watcher` is the current `range element`) and nothing (when others threads are the current `range element`).

Then, the verification expression states that if the set `Called` is not empty, the current `range element` must have a certain period and priority. This expression will be checked against the thread `Watcher`, the only matching component.

6. Conclusion and future work

We considered the problem of requirements enforcement in architectural models prior to performing analysis (schedulability) or code generation. Such enforcement helps making sure the model can be analyzed by these tools, but also allowing one to detect inconsistencies in the model earlier. Yet, the designer needs a precise and concise way to express those, without breaking the link between models and constraints.

We propose REAL, a DSL bound to a model as an AADL annex to express requirements on model entities. REAL defines checks as sets and operations on these sets built from the model. The implementation of a REAL checker allows to assess a model is compliant to a set of structural rules. REAL manipulates AADL component types in an efficient way to check requirements on the model structure. We illustrate how to use REAL for checking restrictions from the Ravenscar profile applied to real-time systems. REAL can also be used by modelers in order to ensure non-functional requirements of some components. Hence, REAL can help building architectural models that respects precise constraints prior to further exploitations.

Our future work will use REAL first to compute metrics on a model to evaluate its performance, by adding formulae to theorems. Connections with other analysis tools will be contemplated to delegate complex computations (e.g. worst-case execution

time) or simulations. Furthermore, we will evaluate model transformation techniques to derive optimized, semantically-equivalent model from the user-defined model. In this context, REAL will provide evaluation of the merit of each model.

7. References

ARINC, *Avionics Application Software Standard Interface, ARINC Specification 653*. January, 1997.

Burns A., Dobbing B., Vardanega T., *Guide for the use of the Ada Ravenscar profile in high integrity systems*. 2003.

Deng W., Dwyer M. B., Hatcliff J., Jung G., Robby, Singh G., « Model-checking Middleware-based Event-driven Real-time Embedded Software », *Proceedings of the First International Symposium on Formal Methods for Components and Objects (FMCO 2002)*, 2003.

Dissaux P., « Using AADL for mission critical software development », *2nd European Congres ERTS (Embedded Reat Time Software*, January, 2004.

Douglass B. P., *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*, Addison-Wesley, SEP, 2002.

Farail P., Gaufillet P., TOPCASED : un environnement de développement Open Source pour les systèmes embarqués, Technical report, Airbus France, 2005. http://www.topcased.org.

Feiler P. H., Gluch D. P., Hudak J. J., The Architecture Analysis & Design Language (AADL): An Introduction, Technical report, CMU, 2006. CMU/SEI-2006-TN-011.

Freitas L., POSIX 1003.21 Standard – Real Time Distributed Systems Communication (in Z/Eves), Technical report, University of York, 2006.

Hugues J., Zalila B., Pautet L., « Rapid Prototyping of Distributed Real-Time Embedded Systems Using the AADL and Ocarina », *Proceedings of the 18th IEEE International Workshop on Rapid System Prototyping (RSP'07)*, IEEE Computer Society Press, Porto Alegre, Brazil, p. 106-112, May, 2007.

Kwon J., Wellings A., King S., « Ravenscar-Java: A High-Integrity Profile for Real-Time Java », *Concurrency and Computation: Practice and Experience*, vol. 17, n° 5-6, p. 681-713, 2005.

Medvidovic N., Taylor R. N., « A Framework for Classifying and Comparing Architecture Description Languages », *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, Springer-Verlag, 1997.

OMG, *UML 2.0 Superstructure Specification*, OMG, 2003.

OMG, *A UML profile for MARTE - Beta 1*, OMG, 2007.

SAE, « Architecture Analysis & Design Language(AS5506) », sep, 2004.

Sha L., Rajkumar R., Lehoczky J., « Priority Inheritance Protocols : An approach to Real-Time Synchronization », *IEEE Transactions on Computers*, 1990.

team S. A., OSATE : an extensible Source AADL tool environment, Technical report, SEI, December, 2004.

Vergnaud T., Zalila B., Ocarina: a Compiler for the AADL, Technical report, Télécom Paris, 2006.

Warmer J., Kleppe A., *The Object Constraint Language : Precise Modeling with UML*, Addison-Wesley, 1998.