Expressing and enforcing user-defined constraints of AADL models

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

Jérôme HUGUES,
Université de Toulouse, ISAE
10, avenue E. Belin - BP 54032,
31055 Toulouse CEDEX 4, France
jerome.hugues@isae.fr

***Abstract*—The Architecture Analysis and Design Language AADL allows one to model complete systems, but also to define specific extensions through property sets and library of models. Yet, it does not define an explicit mechanism to enforce some semantics or consistency checks to ensure property sets are correctly used.**

In this paper, we present REAL (Requirements and Enforcements Analysis Language) as an integrated solution to this issue. REAL is defined as an AADL annex language. It adds the possibility to express constraints as theorems based on set theory to enforce implicit semantics of property sets or AADL models. We illustrate the use of the language on case studies we developed with industrial partners.

I. INTRODUCTION

Model-Driven Engineering is now entering maturity, where developer masters most of its concepts, modeling patterns and artifacts. We see a trends to evolve from small models to explore concepts to large libraries of models to represent product lines.

It is therefore common to build a library of reusable models, each of which comes with some contracts attached. Those contracts can be either "implicit" or "explicit". We call them "explicit" when they rely directly on the underlying meta-models or type systems (syntactic legality rules) to enforce contracts relationships. On the other hand, they are "implicit" when one need to interpret part of the models to understand its full semantics, e.g. based on some naming conventions.

Without lack of generality, we base our work on the AADL [12] language. These concerns are common to most modeling frameworks, should it be UML, SCADE or Simulink. As part of the AADL committee work, some industrial partners indicated that the size of models libraries can be above one million of models elements (component types, property sets, ...).

When building such large models, it is not uncommon to add some implicit information on the models built. However, lack of documentation of some artifacts is sometimes the root cause of severe system's failure like Ariane V maiden's flight [5] or NASA probes failure. Increased size of models is therefore bringing back some common failures of software that were to be minimized thanks to models.

In this paper, we propose to enrich the AADL with an annex language dedicated to checking and enforcing user-specific constraints. We first quickly present the AADL, focusing on user-defined extension mechanisms it provides. We show how these mechanisms may put a risk on models reusability. We then introduce REAL, an AADL annex language whose main goal is to express constraints on models. This language is completed with a checker that validate them. We illustrate the use of REAL through the definition of a property set, and a small library of AADL models.

II. BRIEF OVERVIEW OF AADL

AADL [12] is an architecture description language dedicated to the design of Distributed Real-Time systems standardized by the SAE. AADL is component-centric and allows to specify both software and hardware parts of a systems. It allows one to define consistent block interfaces, and to separate them from block implementation.

An AADL model is made of *components*. Software components (`data`, `thread`, `thread group`, `subprogram`, `process`) are distinguished from execution platform components (`memory`, `bus`, `processor`, `device`) and hybrid components (`system`).

The behavior of a system (e.g. how functional blocks interact) is fully defined in the standard by mean of "properties" (attributes with a dedicated semantics) to progressively refine the semantics of a system, e.g. dispatching invariants, communication patterns; non-functional properties (such as timing, priorities, etc) applied to each model element; non-functional aspects of components can be described within an AADL model such as thread dispatching condition (periodic or sporadic), interface specifications and how components are interconnected. These have a deep impact on the system's behavior. Functional aspects (algorithmic specifications) are attached separately as source code by means of AADL properties. An introduction to AADL can be found in [6].

AADL proposes two interesting extension mechanisms: property sets and `extends`.

Property sets allow one to define custom properties to extend standard ones. This is the path taken by the "Data modeling annex document" [15] that allows one to model

```
Data_Digits : aadlinteger applies to ( data );
-- The Data_Digits property specifies the total number
-- of digits of the fixed point type.
```

Listing 1. A property definition from [15].

precisely data types to be manipulated, or the "ARINC annex document" [13] that defines patterns for modeling ARINC653 systems. Example 1 illustrates one property definition from the "Data modeling annex document". We note that comments statements explicitly state that this property is for fixed point data component types defined through another properties: `Data_Representation`.

AADL `extends` mechanism allows one to reuse and extend one component type. If a component type extends another component type, then features, flows, and property associations can be added to those already inherited. A component type extending another component type can also refine the declaration of inherited feature and flow declarations by more completely specifying partially declared component classifiers and by associating new values with properties. Hence, one can easily derive new models from existing one, following typical derivation as in object-oriented approach a-la UML, or product lines.

Finally, the AADL language can be extended through annex languages. They offer the possibility to attach additional considerations to an AADL component like behavioral specification [14]. Observe that languages may extend the semantics of the AADL core. They must be defined with great care, the definition of the behavioral or error modeling annexes as part of the AS-2C committee shows this is not an easy process .

III. Pitfalls of AADL extension mechanisms

The AADL defines both a full grammar, and additional legality and naming rules to constraint the set of legal models. These rules define consistency checks to ensure the model is meaningful. However, these rules apply only to the properties defined in the core of AADL.

The model designer has limited control over the extensions he may add through property sets or model refinements. The two examples in the previous section illustrate potential pitfalls of large model repositories:

- Inconsistency in use of properties: even if properties are typed, there is no control of coordinated use of properties, and thus AADL legality rules that prevent to model an integer whose codeset is Cyrillic. Yet, codeset is obviously the property of a string or character.
- Inconsistency in the extension of component types. There is no equivalent of the Java's `final` keyword that would prevent the addition of features or properties. Yet, this is an important feature to represent some impossible evolution of a component. Hence, one may want to explicitly forbid the addition of some buses to a processor, or to restrict the bus that can be connected for security or energy reasons. But the AADL language does not support such restrictions.
- Inconsistency introduced by the use of AADL annex languages. Such language allow one to specify one aspect of the system (error modeling, behavioral modeling, ...). Since those can be seen as domain-specific language with their own semantics, it is hard to check they do not violate any semantic element of the AADL standard. Thus, we do not address them.

A mechanism to ensure one does not violate any assumption made on some model elements is to attach constraints that express these assumptions, and define a tool to check them. We take advantage of the AADL annex language mechanism to define a DSL: REAL to define such constraint, and later process them in our AADL toolchain.

In the following, we introduce REAL and then show how it solves the issues we emphasized.

IV. REAL, an AADLv2 annex language

REAL (Requirement Enforcement Analysis Language) aims at checking constraints enforcement on architectural descriptions at the specification step, saving significant time over verification at execution time. In this section, we describe the main features of this language. REAL pursues multiple design goals

- Enabling easy navigation through AADL meta-model elements, yet being at a high-level abstraction. To do so, we discarded the use of OCL and decided to define a specific DSL based on AADL meta-model concepts to ease writing of constraints.
- Allowing to define generic theorems: universal quantifiers ($\forall$, $\exists$) notation is interesting to define theorems that can apply to a wide range of models, not just specific instances.
- Allowing for modularity through definition of separate constraints that can be later combined.
- Being integrated to the AADL as an annex language, so that constraints are coupled to models in the model repository.

From these goals, we defined REAL as follows: REAL is based on set theory and associated mathematical notations. 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 verifies an expression over all the elements of a set that is called the range set.

In order to write complex expressions, one can use predefined sets, which contain the instances of the AADL model of a given type, or build intermediary sets, using relations between elements of sets (e.g. returns the elements of the set A which are subcomponents of any elements of the set B).

Subtheorems calls can be used to extract values computed from range sets different than the current one - thus allowing constructs like *get all the instances of threads which periodicity is equal to the minimum periodicity in the system*. These can also be used to define pre-required constraints on the model.

Finally, subtheorems calls can also be used to build local or global variables, or to check pre-required constraints on the model. Callee theorems inherit during theorem interpretation from the caller environment (the local_set), and the user can pass parameters. Thus, it is possible to design a library of theorems that will be used by higher-level, user-defined theorems.

REAL [7] has been integrated as an annex language in OCARINA [10], our AADL toolsuite. We present full examples of REAL in the next sections and show how it can help enforcing additional semantics checks to AADL models.

V. ENFORCING ADDITIONAL SEMANTICS OF AADL EXTENSIONS USING REAL

In this section, we illustrate some use cases of REAL, demonstrating both language features, but also usefulness to validate implicit constraints arising from AADL models.

A. Property sets

Defining a new property set is a typical activity part of the modeling of complex AADL systems. The rationale is to handle project-specific or process-specific concerns that are not part of the AADL core standard. Typical examples include power consumption, weight analysis, security analysis, ...

Let us consider the property sets defined as part of the "Data Modeling" annex document [15]. This normative document defines a set of properties to define precisely data types, like for instance "an signed integer on 16bits". To do so, a property set has been defined to represent each aspect of such type.

Let us consider the following property:

```
Real_Range: range of aadlreal
  applies to ( data, port, parameter );

  -- Real_Range specifies a range of real values that apply
  -- to the data component. This property is used to
  -- represent real range constraints on data that is of
  -- some real type.
```

Obviously, this property shall be defined only for data types that represent ultimately a real type. Types are defined using the `Data_Representation` property, it defines Array, Boolean, Character, Enum, Float, Fixed, Integer, String, Struct, Union.

A constraint is therefore that the `Real_Range` property shall be applied only if the corresponding data type also has `Data_Representation` set to "float".

Translating as a REAL theorem, we have to check that for all data components, if the property `Real_Range` is defined, then the property `Data_Representation` exists and is set to "float". This is illustrated in the following theorem.

```
--  Real_Range property

theorem check_real_range
  foreach d in data_set do

  --  1/ Check that the "Real_Range" property is applied
  --     only to data type whose representation is float.

  check ((not property_exists (d, "real_range"))
       or (property_exists (d, "data_representation")
        and get_property_value (d, "data_representation") =
                          "float"));
end check_real_range;
```

This theorem is a direct translation of the implication logic connector "$\Rightarrow$" in logic: "$A \Rightarrow B$" is equivalent to "$\neg A \vee B$" using REAL constructs.

Based on this pattern, we may define a full library of theorems to check for the different acceptable combinations. This library is likely to grow quickly. In the context of the data modeling annex, the number of combination to check for is growing linearly with the number of properties in the property sets.

Besides, the cost to evaluate a property is an affine relation of the number of elements in the function to evaluate. Optimized implementation of the internal model representation in OCARINA ([10]) ensures that the cost for evaluating the whole library is limited. This is an important point if one wants to integrate such library at model-time, in an interactive environment like OSATE2 [3].

B. Library of models

Maturity of modeling paradigm and associated framework like UML or AADL is now sufficient to achieve full model reusability. For instance, the LAMBDA [9] or ADAMS [1] project are currently exploring how to model libraries of models. Such projects are a natural consequence of modeling paradigms and supporting tools going mainstream.

As we pointed out, one need to be cautious when defining such models: implicit constraints need to be defined and then enforced.

Let us suppose we want to define a Ravenscar-compatible model of a real-time executive. Generally speaking, the Ravenscar profile defines a set of restrictions applied to a model or a program to enable Rate Monotonic Analysis or Response Time Analysis. This profile [2] 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. This profile has been extended to Real-Time Java, and AADL where it serves in the OCARINA AADL-to-ADA code generator we develop [8].

1) Thread and protected objects restrictions: To be Ravenscar-compliant, an AADL model must comply to several properties. we focus on the three following properties :

- *Tasks are cyclic, with a non null Minimum Inter Arrival Time* : 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 [16]

In some conditions, Ravenscar systems can also comply to the more restrictive rule:

- *RMA-schedulable* : The threads are schedulable according to the Rate Monotonic Analysis

Other restrictions are more specific to the code being used, and cannot be checked at model level (e.g. dynamic task creation, use of delay constructs, ...).

From an AADL model, checked by a set of REAL rules, one can generate code that follows the architectural description with an appropriate code generator like OCARINA. Code patterns would strictly follow architectural patterns. Then verifying code compliance to the Ravenscar profile would have the same complexity as verifying the model compliance to the Ravenscar profile. This earlier check helps in ensuring the quality of the model to requirements. We show how to define the corresponding REAL theorems.

2) Non-aperiodic tasks: The first step toward writing a REAL theorem is to translate the code-related statement in the AADL model. The statement is quite easy to translate : 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`.

In listing 2, `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 2. Task Periodicity

3) PCP-Compliance: The PCP-compliance condition can be divided into two conditions :

- All data components shared by multiples threads have been defined as following PCP
- No data component following PCP has an accessor thread whose priority is superior to the data priority and all those threads must be hosted by the same processor.

The first condition is equivalent to assessing that if more than one thread access a data component instance (i.e. the same data component instance is provided to those threads), then the data component instance must have the `Concurrency_Control_Protocol` property set to `PRIORITY_CEILING`.

The second condition states: all threads accessing a data must have a priority which is less than the priority of the accessed data. Furthermore, all threads must be on the same processor.

The theorem is split in two parts: the theorem in Listing 3 checks whether PCP has been declared for all shared data; the second one (listing 4) checks whether conditions for PCP are present.

We 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 sothat
                           Is_Accessing_To (t, d)}
   check (Cardinal (accessor_threads) = 1 or
          (Get_Property_Value
             (d, "Concurrency_Control_Protocol") =
             "Priority_Ceiling"));
end all_pcp;
```

Listing 3. Shared data access

```
theorem PCP
foreach e in Data_Set do
  accessor_threads := {t in Thread_Set |
                          Is_Accessing_To (t, e)}
  threads_processors := {p in Processor_Set |
                        Is_Bound_To (accessor_threads, p)}

  requires (all_pcp); -- all_pcp theorem is valid

  check (((Get_Property_Value
            (e, "Concurrency_Control_Protocol") <>
           "priority_ceiling") or
     (Get_Property_Value (e, "Priority") >=
           Max (accessor_threads,
                Get_Property_Value, "Priority")))
         and Cardinal (threads_processors) <= 1);
end PCP;
```

Listing 4. PCP

4) RMA Schedulability: A sufficient condition for RMA schedulability for independents threads is defined in [11] by the following formula : $\sum_{i=0}^{N} \frac{C_i}{P_i} < N(2^{\frac{1}{N}} - 1)$, where N is the number of threads in the system, and C_i and P_i are respectively thread i's period and computation time. The value of $\frac{C_i}{P_i}$ is named thread i's utilization (U_i). To represent this factor, we have defined a thread property *Utilization*.

Note that independence is a necessary condition, which means that the threads must not share protected objects. Also, it requires all threads to be periodic, which is a stricter condition that the previous one.

To detect shared protected objects, we use the data

component property *Concurrency_Control_Protocol*. Then, checking that each protected object is accessed at most by one thread ensures independence.

The theorem has been split in two parts. The first, theorem (listing 5), checks whether threads are independents, while the second one (listing 6) checks whether conditions for RMA are met.

```
theorem Independance
foreach e in Data_Set do
  -- actually this set is either one-element or empty
  protected_data :=  {d in Data_Set |
                      Compare_Property_Value
                      (e, "Concurrency_Control_Protocol",
                          "Protected_Access")}
  accessor_threads := {t in Thread_Set |
                       Is_Accessing_To
                         (t, protected_data)}
  check (Cardinal (accessor_threads) <= 1);
end Independance;
```

Listing 5. Independence

```
theorem RMA
  foreach e in Processor_Set do
    Proc_Set(e) := {x in Process_Set |
                    Is_Bound_To (x, e)}
    Threads := {x in Thread_Set |
                Is_Subcomponent_Of (x, Proc_Set)}
    requires (independance);
    check (sum (get_property_value
                (Threads,
                 "RTOS_properties::Utilization")) <=
           (Cardinal (Threads) *
            (2 ** (1 / Cardinal (Threads))) - 1));
end RMA;
```

Listing 6. RMA

: This two case studies illustrate how to define theorems that can be later attached to AADL models to ensure they faithfully comply with some modeling guidelines. We are currently building a library to gather all theorems and make them available through both OCARINA and the OSATE2 modeling environment.

VI. CONCLUSION

The AADL allows one to define models whose semantics goes beyond the standard by adding more information through dedicated property sets or patterns. Yet, we note this comes with limited support from the standard: it is not possible to attach constraints or contracts to particular patterns.

In this paper, we discussed how AADL annex languages can propose a solution, and introduced our annex language REAL. REAL allows the designer to attach constraints to express implied semantics of specific properties or patterns.

REAL relies on simple mathematical notation to define properties: set manipulation and queries on elements of the sets. By doing so, reading or writing theorems is quite natural, without needs to rely on AADL meta-model. Being away from the meta-model roots of AADL was a design goal to achieve higher readability.

REAL has been integrated to our AADL environment OCARINA. It has been successfully used to enforce many constraints derived from the Data Modeling or ARINC653 annex documents [4], but also to represent patterns like POSIX or Ravenscar.

We are now working towards deploying REAL in a more systematic way, to build a comprehensive library of constraints expressed in the AADL core standard and attached annex documents. The objective is to help the user to better understand the model being built and to catch design misconception early in the process.

REFERENCES

[1] ADAMS. Action for the Dissemination and Adoption of the MARTE and related Standards for component based middleware. http://www.adams-project.org/, 2009.

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

[3] CMU/SEI. Open Source AADL Tool Environment (OSATEv2). Technical report, CMU/SEI, 2009.

[4] J. Delange, L. Pautet, and P. Feiler. Validating safety and security requirements for partitioned architectures. In *Ada-Europe '09: Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies*, pages 30–43, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] J. L. L. et al. ARIANE V, FLight 501 Failure. http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.

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

[7] O. Gilles. REAL User's Guide. Technical report, Télécom Paris, 2009. available at http://aadl.enst.fr/real.html.

[8] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite. *ACM Transactions in Embedded Computing Systems (TECS)*, 7(4):1–25, July 2008.

[9] Lambda. LIBRARIES FOR APPLYING MODEL BASED DEVELOPMENT APPROACHES. http://www.usine-logicielle.org/lambda/index_FR.html, 2009.

[10] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues. OCARINA: An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications. In *Reliable Software Technologies'09 - Ada Europe*, volume LNCS, pages 237–250, Brest, France, jun 2009.

[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in hard-real-time environment. In *Journal of the ACM*, january 1973.

[12] SAE/AS2-C. *Architecture Analysis & Design Language v2.0 (AS5506A)*, January 2009.

[13] SAE/AS2-C. *ARINC653 Annex document for the Architecture Analysis & Design Language v2.0 (AS5506A)*, October 2009 2009.

[14] SAE/AS2-C. *Behavioral Annex Language Specification for the Architecture Analysis & Design Language v2.0 (AS5506A) (draft 2.11)*, October 2009 2009.

[15] SAE/AS2-C. *Data Modeling Annex document for the Architecture Analysis & Design Language v2.0 (AS5506A)*, October 2009.

[16] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. In *IEEE Transactions on Computers*, 1990.