Enforcing Software Engineering Tools Interoperability: An Example with AADL Subsets

Vincent Gaudel*, Alain Plantec*, Frank Singhoff*, Jérôme Hugues†, Pierre Dissaux‡, Jérôme Legrand‡
*Lab-STICC UMR 6285, UBO, UEB 20, Avenue Le Gorgeu CS 93837, 29238 Brest Cedex 3, France
Email: {gaudel,singhoff,plantec}@univ-brest.fr
†Université de Toulouse, ISAE 10, Av. E. Belin 31055 Toulouse Cedex 4, France
Email: jerome.hugues@isae.fr
‡Ellidiss Technologies 24, Quai de la Douane, 29200 Brest, France
Email: {pierre.dissaux,jerome.legrand}@ellidiss.com

***Abstract*—Model-Based Engineering is now a valuable asset to design complex real-time systems. Toolchains are assembled to cover the various stages of the process: high-level modeling, analysis and code generation. Yet tools put heterogeneous requirements on models: specific modeling patterns must be respected so that a given analysis is performed. This creates an interoperability paradox: models must be tuned not given system requirements, but to abide to tools capabilities. In this paper, we propose a systematic process to define the definition, comparison and enforcement of tools-specific subsets. Thus, we guide the user in selecting the tools that could support its engineering process. Our contribution is illustrated in the context of the AADL Architecture Design Language.**

***Index Terms*—Architecture Description Language; AADL; Subsets; EXPRESS; Tool Chains; Interoperability;**

I. Introduction

Although Model-Driven Engineering and Architecture Design Languages can greatly help engineers, the development of critical real-time systems remains a difficult task. Designers have to perform architecture design exploration and analysis of their systems all along the design process. This is performed through a careful combination of tools, each of which covers a particular analysis domain. One of the key challenge ahead is to define a sound toolchain, compatible with the constraints set by the industry or the system under construction.

We place our work in the context of the Architecture Analysis & Design Language (AADL) [1]. Yet, the very same issue arises when contemplating other standards or domaines, like OMG SysML or MARTE [2], or AutoSAR [3].

AADL allows to model both hardware and software parts of real-time embedded systems. As an architecture description language, AADL models are composed of interconnected reusable components. An AADL model can be used for multiple purposes such as design, verification, analysis or code generation. There exists various AADL tools covering the full V-cycle: model editors such as OSATE2 [4], ADELE or STOOD, analysis tools like Cheddar [5], but also many mappings from AADL to existing tools like UPPAAL, Petri nets, Timed automata, etc[1] and finally source code generators such as Ocarina [7] or RAMSES[8].

[1]Interested readers may find a complete list of model checking tools for AADL in [6] and http://www.aadl.info

We note that each tool is covering one step in the engineering cycle. Ideally, the designer would create one model that would be refined along all activities. Hence, the same architectural model would be processed by different tools.

From a modeling perspective, the key issue is to guarantee the model is amenable to all those analysis. As a matter of fact, a scheduling analysis tool would accept only a given set of task patterns, security policy enforcement would focus on communication patterns and flow of information whereas fault error modeling and analysis tools would consider other exchange mechanism for fault propagation. Hence one needs to focus on the concepts accepted by a tool or a theory.

We claim this creates an interoperability paradox: models must be tuned to abide to specific tool capabilities or accepted patterns, and not to implement system-specific requirements. To address this issue, we aim at making explicit the subset of the design language that is accepted by a tool, and provide an associated model validation tool. The expected benefits are manyfold: 1) determining early in advance whether a model is compatible with an analysis or a tool, 2) comparing power of expression of subsets accepted by tools, 3) detecting situation where a tool erroneously accepts an incomplete model.

We propose to make explicit tools restrictions and requirements with dedicated models named *Subsets*. Subsets are modeled as logical combination of restrictions upon the AADL meta-model using a Domain Specific Language (DSL in the sequel). These restrictions are used to derive a validation tool to answer the three points we outlined below. In our experiments, all restrictions are specified as cardinality constraints on entities of the AADL meta-model.

This article is organized as follows. Section II briefly introduces AADL and illustrates interoperability issues. Related works are discussed in section III. Section IV discusses the study of three AADL subset examples and the DSL for subsets specification is exposed. The major focus of the section V is to present how our approach was implemented and evaluated, before conclusion in section VI.

II. Overview of AADL

In this section, we first present the AADL language. Then, interoperability issues in tool support are discussed.

A. AADL Architecture Language

AADL (namely Architecture Analysis & Design Language) is an Architecture Description Language that supports the design, the analysis and the integration of distributed real-time systems [1]. The language supports the specification of systems as an assembly of hardware and software components. AADL models integrate functional component interfaces interconnected to non-functional aspects (response time, safety and certification properties, among many others). The interfaces are later refined as component implementations, detailing internals.

An AADL model is made of several categories of software and hardware components. (1) *Software component* categories are threads, processes and sub-programs. A thread component models a schedulable unit of concurrent execution. Process components represent virtual address spaces. A subprogram component models a piece of program that is sequentially run. (2) *Hardware components* specify the execution platform. AADL defines processor, memory, device and bus categories of hardware components. Hardware and software components can be stored in libraries or hierarchical organized in systems. An AADL system component specify how software components are deployed on hardware components.

AADL components interact through component connections. A component connection models data or control flows between components. Connections can model exchange of events or queued messages, rendez-vous or accesses to shared data.

AADL model components can be enriched by properties. A property makes explicit various concepts such as the periodicity of a thread, the name of the source code for a sub-program component, or the available bandwidth for a bus component. Properties are defined in property sets. The AADL standard includes a large set of normalized properties grouped in several standard property sets. Moreover, user defined properties can be used. These kind of application or system specific properties may be added to extend the description with regard to the expected system analysis.

Any AADL model must be compliant with the core AADL language. In addition, a model may embed elements defined in annexes: they either define particular design patterns for a particular domain: data modeling or ARINC653 systems; or a companion language for modeling errors or component behaviors. Companion language elements are attached to model elements and can be ignored by tools if they are outside the considered analysis scope.

Listing 1 is a simple AADL model. It represents a system constituted of two independent threads T1 and T2. T1 is periodic (dispatched periodically) whereas T2 is hybrid (dispatched periodically or triggered by an event). All threads are deployed on a uniprocessor environment. Each thread has its own properties made of a dispatch protocol, a period and a computing execution time. The scheduling policy is *Earliest Deadline First* (EDF).

B. AADL models interoperability issues

We note several AADL processing tools exist, covering many V&V activities. From this list, designers could tailor their toolchain for the construction of critical real-time systems. In these toolchains, AADL is used as a central language to store all architectural artifacts from which analysis model (e.g. a Petri Net, a fault-tree) can be derived.

```
package Example
public
system S end S;
system implementation S.i
subcomponents
   A1 : process A.i;
   P1 : processor P.i;
properties
   Actual_Processor_Binding => (reference(P1) applies to A1;
   Scheduling_Protocol => (EARLIEST_DEADLINE_FIRST_PROTOCOL)
         applies to P1;
end S.i;
processor P end P;
processor implementation P.i end P.i;
process A end A;
process implementation A.i
subcomponents
   T1 : thread T.i;
   T2 : thread TT.i;
end A.i;
thread T
properties
   Dispatch_Protocol => Periodic;
   Period => 10 ms;
   Compute_Execution_Time => 5 ms..5 ms;
end T;
thread implementation T.i end T.i;
thread TT
properties
   Dispatch_Protocol => Hybrid;
   Period => 25 ms;
   Compute_Execution_Time => 3 ms..4 ms;
end TT;
thread implementation TT.i end TT.i;
end Example;
```

Fig. 1. Modeling a system with AADL.

The richness of AADL is, in retrospect, the root cause of key interoperability issues. These are derived from the power of expression of the language, that induce heterogeneous modeling patterns and thus make tool support more complex. Let us illustrate this claim:

a) Power of expression of AADL: AADL is a rich architecture language for modeling a large palette of architectures and systems. As an architecture description language, it allows the description of structural aspect of both software and hardware components of an architecture. Behavior part is added in different ways: properties, connections between elements, binding relations allocating execution resources to components and annexes. This separation of concerns is compatible with a declarative way of modeling, yet the full characterization requires a full understanding of all AADL concepts.

This is well embodied by the size of its meta-model, describing all its concepts. The full AADLv2 meta-model has more than 100 elements. As another illustration, the AADL BNF has 185 syntax rules. The AADL core language standard describes around 250 legacy rules and more than 500 semantic rules. This makes AADL quite a rich language.

b) Modeling patterns: Feiler et al. identify two ways to use AADL for analysis [9]: lightweight analysis of architecture patterns to discover systemic problems and full scale analysis based on theoretical frameworks like sensitivity analysis, schedulability or reliability analysis, code generation, etc.

Yet, given the richness of AADL, similar information may be presented differently. This diversity in modeling patterns may stem from the expertise of the team, the number of refinements performed or the analysis objectives.

c) Impact on tool supports: AADL tools rely on typical model-based transformation engines to map an AADL model onto an analytic model. These tools implement pattern recognition strategies to map relevant concerns onto an intermediate model that convey only the abstraction required for a given analysis such as a task set for scheduling analysis.

Hence, if a tool supports only a restricted set of patterns, or conversely it a model makes use of convoluted one, the mapping may be incomplete. This could lead to inaccurate analysis model, hence wrong results. More problematic, if several tools are to be used in conjunction, they should rely on similar patterns to maximize efficiency.

This is what we call the *interoperability issue*: AADL models must be designed according to the known tool restrictions. As a consequence, deciding how to design a particular system with AADL strongly depends on the tools that will be used. Let us note this issue is not specific to AADL, it plagues all model based software engineering toolchains that rely on a large modeling framework built on AADL, but also MARTE [2] or AUTOSAR [3].

C. Contributions to support AADL tool interoperability

In order to ease AADL toolchains designing, we propose to make explicit the subset of AADL a tool support using a dedicated model. Such model describes the subset of the AADL language it supports in terms of constructs and modeling patterns. Then, a validation process checks whether a model is compatible with this subset.

This model serves two complementary purposes:

- One would know precisely whether a model is “ready” for a given analysis, that is all information is readily available in the model; or on the contrary some violations have been detected;
- Furthermore, by combining subsets, one would extend the range of analysis that can be supported by a model.

III. Related Works

Interoperability between software engineering tools and language subsets have been subject to numerous studies. We note there are two main ways to enforce interoperability between software engineering tools: 1) model transformations between tool specific models or 2) the definition of proper subsets.

Many approaches with model transformations have been investigated. For example, Garlan et al. [10] propose a language to specify transformations between pairs of tools. This approach implies that the consistency of data between the different pairs of tools has to be ensured, which may be difficult when tools are updated. Malavolta et al. propose DUALLy [11], a framework for model-to-model transformations based on a high level meta-model that maps semantically equivalent elements between the two models. Finally, the AMMA model engineering platform offers bijective transformations from tools meta-models toward a pivot meta-model [12].

Unfortunately, those examples do not cope with the completeness issue of interoperability between tools. Indeed, being able to map similar concepts within different models does not ensure that a model provides required information.

Definition of language subsets has also been extensively investigated. They aim at restricting the scope of a programming/modeling language. For example, Burns *et al.* define Ravenscar, a subset for the Ada general purpose programming language [13]. This Ada subset allows programmers to write Ada programs that are compliant with real-time critical systems requirements. Delange et al. [14] define guidelines for the implementation of ARINC653 systems using AADL. This ARINC653 guidelines specifies a subset of AADL to model all the concepts of an ARINC 653 architecture. Finally, a set of restrictions for the scheduling analysis with MAST [15] has been proposed. Those restrictions are written as a set of rules expressed using natural language. Each of these examples propose a subset with various languages, which make use of them to support tool interoperability a difficult task.

In our approach, we express all subsets from a standard pivot language and with a restricted DSL, which allow us to compare subsets supported by tools and help supporting tool interoperability.

IV. What do we need to model AADL Subsets

In this section, we explain why AADL subsets can be modeled as sets of cardinality constraints.

An important is that we consider that the primary goal of a subset is to specify which ”kind of AADL model” is use in a given context. Then, a subset is made for a particular utilization.

First we present three subsets proposed by AADL tool designers. Based on those three subsets defined as sets of restrictions, we discuss their differences and identify the different kinds of used restrictions. Then, we explain why and how those different kinds of restrictions can be normalized as cardinality constraints.

A. Examples of AADL subsets

Here, we present the subsets for the AADL tools (1) Marzhin [16], (2) BLESS [17] and (3) Cheddar [18]. Those three tools are respectively dedicated to scheduling simulation (1), formal verification (2) and schedulability analysis (3). For space and clarity reasons, we only provide parts of the restrictions used to defined each subsets. We also present each restriction with an unique identifier.

1) Subset of Marzhin: Marzhin is a simulation tool for AADL models. It is integrated to AADLInspector. The intention of this subset is to ensure that an AADL model can be simulated with Marzhin 1.0. Its subset is composed of twelve restrictions. The restrictions are related to three property sets: properties defined in the AADL_Project property set, the ARINC653 property set and some properties and in the Cheddar Property

Set [19]. They also restrict the use of the Behavior annex to a limited set of behavioral operators. The restrictions for the Marzhin tool are the following:

MA1 There is only one processor component.
MA2 The property Actual_Processor_Binding must be specified.
MA3 For all processors, property Scheduling_Protocol must have one of the following values only: POSIX_Fixed_Priority_Scheduling_Protocol, Rate_Monotonic_Protocol or Deadline_Monotonic_Protocol.
MA4 The property Dispatch_Protocol must have one of the following values only : Periodic, Aperiodic, Timed, Hybrid, Background.
MA5 Only the following properties are allowed : Dispatch_Protocol, Period, Deadline, Priority and Compute_Execution_Time.
MA6 Features must be one of the following : Event_Port, Event_Data_Port, Provides_Subprogram_Access, Requires_Subprogram_Access, Requires_Data_Access
MA.. ...

2) The BLESS subset: The BLESS tool assumes that any AADL model to be analysed must be compliant to a subset issued from 'AADL-light'. The intention of AADL-Light is to capture most commonly used parts of AADL for beginners. It forbids the use of complete aspects of AADL (as sub-program calls for instance). This subset is modeled by eleven constraints:

BL1 There is no flows.
BL2 There is no inheritance.
BL3 There is no refinement.
BL4 There is no subprogram call sequences.
BL5 There is no contained property association.
BL.. ...

3) The Cheddar subsets: In [18] Gaudel et al. proposed five architectural design patterns. Each of these design patterns is an AADL subset. The intention of these subsets is to ensure that an AADL model can be analysed with the schedulability analysis methods implemented into Cheddar. Indeed, to be analysable with Cheddar, an AADL model must be compliant with one of these subsets. In this article, we present one of these subsets called "Time-Triggered". This subset is specific to systems using time-triggered communications *a la* Meta-H. In AADL, such communications are achieved through AADL data ports. Each AADL thread reads its inputs data ports at dispatch time and writes its output data ports at completion time. This subset is composed of 10 constraints:

TT1 There is only one processor component.
TT2 All threads must be periodic.
TT3 All connections must be data port connections.
TT4 There is no data component.
TT5 All features must be data Port.
TT6 There is at least one port connection.
TT7 For all data port, property Timing must have one of the following values only: *sampled*, *immediate* or *delayed*.
TT8 For all processor components, property Scheduling_Protocol must be specified and must have the following values only: POSIX_Fixed_Priority_Scheduling, Rate_Monotonic, Earliest_Deadline_First or Deadline_Monotonic.
TT.. ...

The example presented in section II is compliant with the Marzhin subset but it is incompatible with Cheddar schedulability analysis associated to the Time-Triggered subset. Indeed, the thread T2 is not periodic, and there is no time-triggered communication (modeled as a connection between two data ports in AADL). This shows that there may be two kinds of interoperability issues: the absence of a required element, or the presence of a forbidden one.

B. What do we need to properly define subsets

From the examples explained previously, in this section we outline what is required to define subsets.

In documentations, restrictions are expressed using natural languages. Obviously, this can lead to serious interpretation and implementation issues because the same constraint can be expressed differently. More, we expect that subsets of different tools of a given tool-chain can be compared in order to allow the specification of a subset for the whole tool-chain. For instance, restrictions MA4 and TT1 both restrict the temporal behavior of threads dispatch. MA4 gives a set of potential values for a property, whereas TT1 requires that threads have a given behavior. One could also notice that if an AADL model is compliant with TT1, it implies that it is also compliant with MA4. This is a crucial information to ensure that a tool requiring TT1 could also satisfy MA4 for any other tool in order to ensure that they could be jointly used in a tool-chain. To allow the precise specification and the comparison of subsets we propose a dedicated language. Our proposal is based on the observations that we made over the examples given previously.

First, we observed that subsets can differ in their objectives and do not restrict the same things. The subset of Marzhin and the subset of Cheddar restrict AADL models as instantiated AADL components. Whereas the BLESS subset forbids some of the AADL language constructs (inheritance, refinement,...), and thus restricts the declarative model of AADL. AADL component instances can be deduced from the declarative model instances (each component instance is a *subcomponent* of a *system component*). As a consequence, we expect that all restrictions must restrain the AADL declarative meta-model.

Second, we observed that there are two kinds of restrictions:

- Some restrictions forbid or require the presence of a given AADL element. We call them *global restrictions* because they need to be met within the scope of the entire AADL model (i.e. restrictions TT3 for instance).
- Other restrictions forbid or require a particular value for a given AADL element attribute or consists on a constraint over the relationship between two AADL entities. We call them *local restrictions* (i.e. restriction MA4 for instance).

In order to express these two kinds of constraints in a similar way, we propose to specify them as cardinality constraints over the AADL meta-model. Global constraints are naturally cardinality constraints since they require the presence or absence of a given AADL element. Local constraints restrict relationships between several AADL entities or between an AADL element and one or several of its attributes. Thus, also in the case of a local constraint, a cardinality constraint requires or forbids the presence of a given relationship.

C. A domain specific language to define subsets constraints

To allow the clear specification of subsets in a concise and homogeneous way we developed a domain specific language (DSL in the sequel). A specification consists in the declaration of a subset by means of a name associated with all the constraints that an AADL model must meet to be considered as compliant with the subset. This section describes our DSL by example.

1) Declaring a subset: The *Subset* keyword is used followed by the name of the subset. All constraints are then specified between parenthesis. The following example declares the Marzhin subset:

```
Subset Marzhin ( ... )
```

2) Adding a global constraint: A constraint is made of a name and of an expression separated by ':'. In the following example, the Marzhin *MA1* global constraint is added. To be compliant with this constraint, an AADL model must have one and only one Processor:

```
Subset Marzhin (
  MA1 : There must be exactly 1 Processor;
...)
```

This form of constraint restricts the cardinality of a set of AADL elements within an AADL model, e.g. *MA1* constraints the size of the set of all processors to be exactly one. Three cases of simple cardinality restrictions are used: 'at most', 'at least' and 'exactly'. For instance, the constraint *TT5* is expressed as:

```
Subset Time-Triggered (
  ...
  TT5 : There must be at least 1 Port_Connection;
...)
```

As a third example of a global constraint, the Time-Triggered *TT4* constraint specifies that features can only be data ports:

```
Subset Time-Triggered (
  ...
  TT4 : Feature must be Data_Port;
...)
```

This kind of constraint specify that all elements that has *Feature* as one of its type must have *Data_Port* as one of its other types. All usable types are the ones of the AADL meta-model elements as they are listed in the appendix C of the AADL standard.

The logical binary operators *AND* and *OR* can be used to specify a constraint expression. Hence the Marzhin *MA6* constraint is declared as:

```
Subset Marzhin (
  ...
  MA6 : ( ( ( ( Feature must be Event_Port
        OR Feature must be Event_Data_Port )
        OR Feature must be Provides_Subprogram_Access )
        OR Feature must be Requires_Subprogram_Access )
        OR Feature must be Requires_Data_Access );
...)
```

3) Adding a local constraint: A local constraint is added the same way as for a global constraint. It is expressed for a particular type (*Thread* in TT2 and *Processor* in MA2 in the example below). The associated rule constrains only one of its attributes (the properties *Dispatch_Protocol* and *Actual_Processor_Binding* in, respectively, TT2 and MA2). A property can be constrained to exist (see MA2) or further, to be specified with a particular value (see TT2):

```
Subset Time-Triggered (
  ...
  TT2 : For all T in Thread : ( the value of
        T.Dispatch_Protocol must be Periodic ); ...)
Subset MARZHIN (
  ...
  MA2 : For all P in Processor :
    ( P.Actual_Processor_Binding must be Specified ); ...)
```

A local constraint can also be used to restrict the declarative model of AADL. The following example forbids inheritance of AADL component classifier as it is required by the subset BLESS.

```
Subset BLESS (
  ...
  BL2 : For all C in Component_Classifier :
    ( C.inherits must be empty );
...)
```

D. How we compare AADL Subsets

In the previous section, we illustrated our DSL for subset definition. This section is dedicated to the definition of relationships between subsets. We define four concepts: between two subsets, *inclusion of a subset by another*, *incompatibility between two subsets*, *intersection of two subsets*.

A subset *A* and a subset *B* are *equivalent* if and only if *A* and *B* are specified by the same constraints. Equivalence between the subsets of two tools means that they are fully interoperable.

A subset *A* is *included* in a subset *B* if for all constraint c_B specifying *B*, there exists a constraint c_A specifying the subset *A* such that the respect of c_A implies the respect of c_B. Thus, any AADL model compliant with the subset *A* will be compliant with the subset *B*. The inclusion of a tool's subset in another tool's one means that the latter can handle all AADL models the first tool can handle.

A subset *A* and a subset *B* are *incompatible* if there is a constraint c_B specifying *B*, there exists a constraint c_A specifying the subset *A* such that the respect of c_A implies the non-respect of c_B. Thus, any AADL model compliant to the subset *A* will not be compliant to the subset *B*. An incompatibility between two tools' subsets implies that the tools manipulate disjoints sets of AADL models and are not able to be part of the same toolchain.

The subset *I* is the intersection of a subset *A* and a subset *B* if for all constraint c_I specifying *I*, there exists a constraint c_{AorB}

specifying the subset *A* or the subset *B*. Thus, the intersection of *A* and *B* is a subset of both of them. The subset of a toolchain is the intersection of the subsets of all tools composing it.

V. Implementation and Evaluation

In the previous section, we proposed a DSL for the definition of AADL subsets. In this section, we investigate if this DSL is well suited to existing AADL tools. For such purpose, we propose a method for subset comparison. This method enables to show incompatibilities among toolchains. We build a AADL meta-model and we model several subsets from AADL tool-designers. The AADL meta-model and its AADL subsets are designed with EXPRESS, an ISO data modeling language devoted to tool interoperability. We conclude with a short discussion on this evaluation.

A. Studying toolchains' subsets

Subsets can be compared using their relationships (see section IV-D). For a system model to be supported by all tools within a toolchain, it has to be compliant with the intersection (called subset$_{tc}$) of all the tools' subsets. Thus, it has to met the intersection of all constraints of all tools' subsets.

Within subset$_{tc}$, there are four kinds of relationships between two constraints:

- identical constraints (e.g. *TT1* and *MA1*)
- one constraint c_1 implies one other c_2 (c_2 is an *over-restricted* constraint) (e.g. *TT2* implies *MA4*)
- one constraint implies the non-respect of one other (incompatible constraints) (e.g. *TT5* and *MA6* are incompatible)
- independent constraints (e.g. *BL2* and *MA1*)

The first tree relationships needs to be explicited (constraints are independent by default). For all the tools to be interoperable, two constraints within subset$_{tc}$ can not be incompatible. If there are some incompatible constraints, tools need to be modified in order to release at least one of the two constraints.

The final form of subset$_{tc}$ is the intersection of all constraints of all subsets, minus all constraint double and over-restricted constraint. For instance, the subset intersection of both Marzhin and Cheddar Timed-triggered will only contain the DSL expression of TT1 and won't contain MA4.

B. Superset modeling and subsets specification using EXPRESS

EXPRESS is a data modeling language [20] which has been developed as a normalized part of the STEP standards [21]. A data model is made of a set of named schemas. A schema can reuse other schemas. It contains primary modeling elements which are constants, types, entities, procedures, functions and global rules. Entities are used to specify domain concepts. An entity contains a list of attributes that provide buckets to store meta-data while local constraints are used to ensure meta-data soundness. Local constraints can be of four kinds: (1) the unique constraint allows entity attributes to be constrained to be unique either solely or jointly, (2) the derive clause is used to represent computed attributes, (3) the where clause of an entity constraints each instance of an entity individually and (4) the inverse clause is used to specify the inverse cardinality constraints. Entities may inherit attributes from their supertypes. As an example, Figure 2 shows an excerpt of our AADL meta-model.

```
SCHEMA AADL_Named_Elements;
 USE FROM AADL_Properties;
 USE FROM AADL_Components;

 ENTITY Named_Element;
  Name :  STRING;
  Properties  :  LIST OF Property;
 END_ENTITY;

 ENTITY Classifier  SUBTYPE OF (  Named_Element ) ;
 END_ENTITY;

 ENTITY Component_Classifier  SUBTYPE OF (Classifier,
      Component) ;
   Inherits  :  OPTIONAL Component_Classifier;
 END_ENTITY;  . . .
END_SCHEMA;
```

Fig. 2. Excerpt of the AADL meta-model specified using EXPRESS.

With EXPRESS, a way to ensure meta-data soundness is to use global rules. Global rules can be added in order to precise data semantic in a given context. Figure 3 shows the example of two global rules *Only_One_Processor* and *Actual_Processor_Binding_Must_Be_Specified*.

```
SCHEMA Subset_Marzhin;
  RULE Only_One_Processor FOR (  Processor ) ;
  WHERE
    MA1 :  SIZEOF (  Processor  )  = 1 ;
  END_RULE;

  RULE Actual_Processor_Binding_Must_Be_Specified  FOR (
       Property  ) ;
  WHERE
    MA2 :  SIZEOF (  QUERY (  p <* Property  |
      (  p. Property_Name  =  'Actual_Processor_Binding'  )  )  )
           > 0 ;
  END_RULE;
END_SCHEMA;
```

Fig. 3. Examples of EXPRESS global rules

For each construct of the DSL, we provide its translation in EXPRESS, in such a way that for each `constraint` the built rule is expressed as a cardinality constraint. A description of the translation from the DSL to EXPRESS can be found here [22]. We used Platypus to elaborate the AADL meta-model and specify subsets [23]. Platypus is a meta-environment based on ISO STEP technology. It enables to design, to verify and to validate meta-models written with EXPRESS.

C. Discussion on the evaluation

In order to evaluate our approach, we modeled several subsets provided by AADL tool-designers and collected within the literature. We specified subsets for BLESS (*BL*), MARZHIN 1.0 (*MA*), Cheddar Time-Triggered (*CTT*), Cheddar Ravenscar (*CR*) and Cheddar Unplugged (*CU*). For each of those subsets, we were able to express them using our DSL and also to translate them toward EXPRESS. The figure 4 gives some metrics about the implemented subsets. The diagonal contains the number of constraints in each subset. For each couple of subsets: the numbers in bold (upper half) count **identical constraints + over-restricted constraints**, and numbers using *this font* (lower half) count incompatible constraints. One can note that identical constraints are numerous for Cheddar Subsets. Indeed, architectural design patterns share a set of five constraints restricting the hardware part of AADL models,

Subsets	*BL*	*MA*	*CTT*	*CR*	*CU*
BL	11	**0+1**	**0+4**	**0+3**	**2+2**
MA	*0*	12	**2+2**	**2+4**	**2+6**
CTT	*0*	*1*	10	**5+1**	**5+1**
CR	*1*	*1*	*3*	13	**6+1**
CU	*0*	*0*	*1*	*1*	12

Fig. 4. Tabular summing up metrics of implemented subsets.

i.e. they define the execution platform for which the patterns have been designed.

This evaluation showed that we are able to model and compare a representative group of subsets. Second, this modeling of subsets helps to identify interoperability failures that are not suspected a-priori.

The complete specification of each subset (with the DSL and EXPRESS), the AADL meta-model, a more complete definition of the DSL are provided here [22].

VI. Conclusion and Future Works

The development of critical real-time embedded systems is a difficult task. There exists numerous tools for Model Based Engineering of such systems. Yet, tools are usually devoted to a particular use and have specific requirements and restrictions. This leads to interoperability failures between them. We propose an approach to tackle interoperability issues by the modeling of restrictions and requirements of each tool with dedicated models named subsets. A subset is defined as a list of cardinality constraints on a ADL meta-model. We illustrate our approach with AADL, a standard language to design real-time embedded systems. We designed several subsets from restrictions and requirements of several AADL tools. For such a purpose, we have proposed a DSL for the specification those AADL subsets. By the implementation of several verification and subset compliance checkers, we have shown that this DSL is well suited for the specification of tool capabilities and restrictions. This work enables to en-light incompatibilities between tools. This may help tool designers to identify missing features of their tools and to improve them. Software engineering tools are often difficult be used. Especially, it is difficult for users to know if their models are compliant with a given tool. We can also expect that this work may help them to automatically check compliance of their models with tool restrictions and also to show them how their models can be adapted according to tools requirements. In future works, we would like to investigate how to automatically adapt user models according to tools requirements. This work will be presented to the AADL standardization committee in order to be introduced to the AADL standard as an annex devoted to AADL tools interoperability.

VII. Acknowledgments

This project is supported by Ellidiss Technologies and Conseil Régional de Bretagne.

References

[1] P. Feiler, B. Lewis, and S. Vestal, "The SAE AADL standard: A basis for model-based architecture-driven embedded systems engineering," in *Workshop on Model-Driven Embedded Systems*, May 2003.

[2] Object Management Group, "MARTE specification," 2005.

[3] S. Bunzel, "Autosar–the standardized software architecture," *Informatik-Spektrum*, vol. 34, no. 1, pp. 79–83, 2011.

[4] P. H. Feiler and A. Greenhouse, "Osate plug-in development guide," *CMU. Pittsburgh*, 2006.

[5] P. Dissaux and F. Singhoff, "Stood and cheddar : Aadl as a pivot language for analysing performances of real time architectures," jan 2008.

[6] J. Hugues, "Analytic virtual integration of cyber-physical systems & AADL: challenges, threats and opportunities," in *Proceedings of the second Analytic Virtual Integration of Cyber-Physical Systems Workshop*, (Vienna, Austria), Nov. 2011.

[7] 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 on Embedded Computing Systems (TECS)*, vol. 7, no. 4, p. 42, 2008.

[8] F. Cadoret, E. Borde, S. Gardoll, and L. Pautet, "Design patterns for rule-based refinement of safety critical embedded systems models," in *Engineering of Complex Computer Systems (ICECCS), 2012 17th International Conference on*, pp. 67–76, 2012.

[9] P. Feiler, "Embedded system architecture analysis using sae aadl," tech. rep., DTIC Document, 2004.

[10] D. Garlan, R. T. Monroe, and D. Wile, "Acme: Architectural description of component-based systems," *Foundations of component-based systems*, vol. 68, pp. 47–68, 2000.

[11] I. Malavolta, H. Muccini, P. Pelliccione, and D. A. Tamburri, "Providing architectural languages and tools interoperability through model transformation technologies," *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, pp. 119–140, 2010.

[12] J. Bézivin, H. Bruneliere, F. Jouault, and I. Kurtev, "Model engineering support for tool interoperability," in *Proceedings of the 4th Workshop in Software Model Engineering (WiSME 2005), Montego Bay, Jamaica*, vol. 2, 2005.

[13] A. Burns, B. Dobbing, and G. Romanski, "The Ravenscar tasking profile for high integrity real-time programs," in *Reliable Software TechnologiesAda-Europe*, pp. 263–275, Springer, 1998.

[14] J. Delange, L. Pautet, A. Plantec, M. Kerboeuf, F. Singhoff, and F. Kordon, "Validate, simulate, and implement arinc653 systems using the aadl," in *ACM SIGAda Ada Letters*, vol. 29, pp. 31–44, ACM, 2009.

[15] M. G. Harbour, J. G. Garcia, J. Palencia, and J. Drake Moyano, "MAST: modeling and analysis suite for real-time applications," in *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pp. 125–134, IEEE Comput. Soc, 2001.

[16] Ellidiss web site. http://www.ellidiss.fr.

[17] B. Larson, P. Chalin, and J. Hatcliff, "Bless: Formal specification and verification of behaviors for embedded systems with software," in *NASA Formal Methods Symposium, SNFM 2013* (N. R. G. Brat and A. Venet, eds.), vol. 7871 of *Lecture Notes in Computer Science*.

[18] V. Gaudel, F. Singhoff, A. Plantec, S. Rubini, P. Dissaux, and J. Legrand, "An ada design pattern recognition tool for aadl performance analysis," *Ada Lett.*, vol. 31, pp. 61–68, Nov. 2011.

[19] F. Singhoff, "The cheddar aadl property sets (release 2.x)," tech. rep., LISyC Technical report number singhoff-03-2007, February 200.

[20] ISO TC184/SC4/WG11 N041 WD, *EXPRESS Language Reference Manual*, 1997.

[21] ISO 10303-1, *STEP Part 1: Overview and fundamental principles*, 1994.

[22] Cheddar Project Web Site. http://beru.univ-brest.fr/svn/CHEDDAR/docs/AADLSubsets, 2013.

[23] A. Plantec and V. Ribaud, "PLATYPUS : A STEP-based Integration Framework," in *14th Interdisciplinary Information Management Talks (IDIMT-2006)*, (République Tchèque), pp. 261–274, Sept. 2006.