Analysis as a first-class citizen: an application to Architecture Description Languages

Jérôme Hugues
Université de Toulouse, ISAE
31055 Toulouse, France
jerome.hugues@isae.fr

Guillaume Brau
University of Luxembourg, LASSY
L-1359 Luxembourg, Luxembourg
guillaume.brau@uni.lu

***Abstract*—Architecture Description Languages (ADLs) support modeling and analysis of systems through models transformation and exploration. Various contributions made proposals to bring verification capabilities to designers through model-based frameworks and illustrated benefits to the overall system quality.**

Model-level analyses are usually performed as an exogenous, unidirectional and semantically weak transformation towards a third-party model. We claim such process can be incomplete and/or inefficient because gathered results lead to evolution of the primary model. This is particularly problematic for the design of Distributed Real-Time Embedded (DRE) systems that has to tackle many concerns like time, security or safety.

In this paper, we argue why analysis should no longer be considered as a side step in the design process but, rather, should be embedded as a first-class citizen in the model itself. We review several standardized architecture description languages, which consider analysis as a goal. As an element of solution, we introduce current work on the definition of a language dedicated to the analysis of models within the scope of one particular ADL, namely the Architecture Analysis and Design Language (AADL).

I. Introduction

Distributed Real-time Embedded (DRE) systems are integral part of safety-critical domains such as transportation, telecommunications, health services, military or space. These systems have to meet both functional and non-functional requirements. For this, DRE systems encompass specific technologies well-beyond software engineering to realize the required service with the expected performances (e.g. time, security or safety) through dedicated networks, processors and real-time operating systems. Prior to system exploitation, the Engineering process has to ensure the system will correctly behave, that is, functional and non-functional expectations are met.

Engineering practices address the development life-cycle of DRE systems following successive and precise activities spanning from the definition of users' requirements, via system design and implementation, to final V&V (Verification & Validation) activities. Yet, many experiments indicate that the distance between the activities steps is detrimental and usually slows down the development process [1]. In practice, a significant part of errors is injected at early-stages of the engineering process, while being detected later in the process. As a consequence, regressions and rework activities have an important weight on the overall project costs.

As part of the solution, model-based methodologies have emerged to support and alleviate traditional practices. Methodologies proposed in the context of Model-Driven Engineering (MDE) involve models definitions and transformations to cover the development life-cycle towards system qualification. It has been successfully applied in two dimensions: 1) definition of standardized notations for the modeling of complex systems architectures such as AUTOSAR/EAST-ADL [2], [3], OMG MARTE and SysML [4], [5] or SAE AADL [6] and 2) model-based support for V&V activities, targeting model checkers, performance analysis tools, etc.

Merging model-based solutions with the typical V-cycle leads to the manipulation of various models capturing systems elements, from early requirements down to implementation. During the *design* phases, models are refined from abstract to more precise ones; traceability links can be maintained to link design and implementation choices to earlier requirements. In complement, part of *verification* activities may be carried out thought analyses applied on the model(s) so as to ensure that the designed system is sound; checking components interfaces, verifying the non-functional properties are met and preserved across modeling activities are among those analyses. Although such approaches may favor the early discovery of errors, design and verification activities remain loosely coupled and the use of analyses in the process not clearly stated so far. Still, previously underlined drawbacks persist.

We claim that modeling and analysis should be jointly carried out within a common framework. More importantly, analysis – as a set of model assessment activities – should be part of the model in a form that constraints its construction.

In Section II, we review several existing MDE frameworks dedicated to the architecture design and analysis of DRE systems and sketch some elements of comparison. The Section III outlines several challenges that show why analysis should no longer be considered apart from the design process but, rather, should be involved for the definition of models and their evaluation. In the last part of the paper (section IV), we provide some elements of solution applied to the Architecture Analysis and Design Language (AADL) and introduce the AADL Constraint Language that aims at unifying AADL modeling concepts and analyses.

II. Analysis in the scope of Architecture Modeling Frameworks

The term "analysis" has several definitions. In this paper, we retain the following[1]: *"a careful study of something to learn about its parts, what they do, and how they are related to each other; an explanation of the nature and meaning of something"*. Analysis is concerned both by the architecture of a system, and the information or knowledge we gain from it.

Analysis is a central modeling objective in MDE, yet with different supporting approaches. In the following, we review support for analysis in OMG MARTE and SysML, AUTOSAR/EAST-ADL and SAE AADL.

A. MARTE and SysML

MARTE (Modeling and Analysis of Real-Time and Embedded Systems) [4] is a UML profile for the modeling and analysis of real-time and embedded systems. It relies on domain-specific extensions of general UML to bring concepts for modeling real-time and embedded applications. These extensions focus on non-functional elements of real-time applications. These elements fall in two categories: quantitative and qualitative aspects at different levels of abstraction. Finally, they may be defined to support modeling, analysis, or both.

MARTE is structured as a hierarchy of UML profiles. The top package, which is the foundation of MARTE, consists of four UML extensions (sub-profiles):

Non-functional properties (NFP): this profile provides modeling constructs for declaring, qualifying, and applying semantically well-formed non-functional aspects of UML models as data types. It is complemented by VSL, the Value Specification Language. VSL is a textual language for specifying algebraic expressions, stating relationships between types, etc.

Time: defines concepts for time in various representations and models: chronometric, logical and synchronous.

Resource (GRM): this profile proposes definition of resources used by software and associated resource usage. It is an abstract level of modeling.

Allocation modeling (Alloc): this profile allows designers to allocate functions to entities to support it, e.g. allocation for scheduling. Non-functional characteristics may be attached to an allocation description (e.g., when specifying the allocation of a function to a given execution engine, it is possible to specify its worst case execution time).

Model-based analysis using MARTE is done through the extensions defined either in the Generic Quantitative Analysis Modeling profile (GQAM), or one of its refinements, dedicated to schedulability analysis and performance analysis. The annotation mechanism used in MARTE to support model-based analyses uses UML stereotypes. These typically map the UML model elements of the application into corresponding analysis domain concepts, and also allow specification of values for properties that are needed to carry out the analyses.

[1]coming from http://www.merriam-webster.com/

MARTE is associated with modeling process to guide the designer, like Optimum [7] which clarifies usage of MARTE concepts for schedulability analysis, or dependability [8]

Let us note there is an on-going work to define a convergence of NFP concepts of MARTE with SysML [], and define a seamless interface between system engineering concerns and precise engineering of embedded systems.

B. AUTOSAR and EAST-ADL

AUTOSAR (AUTomotive Open System ARchitecture) [2] is a standardized automotive software architecture, developed as a cooperation between automobile manufacturers, suppliers and tool developers. Its objective is to create and establish open standards for automotive electronic architectures.

AUTOSAR follows a component-based approach. A model relies on application software components that are linked through an abstract component: the virtual function bus.

This bus connects the different software components in the design model. This abstract component interconnects the different application software components and handles the information exchange between them. This bus abstracts away all hardware and system services offered by the vehicular system, allowing designers to focus first on the function, and later on the actual hardware/software decomposition.

In addition to this component model, AUTOSAR specifies standardized interfaces for all the application software components necessary for various automotive applications.

AUTOSAR enforces a standardized modeling process, using a layered architecture to preserve separation of concerns:

- Basic Software Layer: this layer acts as an hardware abstraction layer;
- Runtime environment: handles the information exchange between software components bound to various Electronic Control Units (ECUs), it has the role of a communication middleware;
- Application Layer: hosts the actual functions.

Both the Basic Software and Runtime Environment implement the virtual function bus of AUTOSAR.

As defined, AUTOSAR is focused primarily on the implementation of the ECUs through system description, configuration and generation of software binaries. It does not address higher-level requirements.

EAST-ADL (Electronics Architecture and Software Technology - Architecture Description Language) [3] is an Architecture Description Language (ADL) for automotive embedded systems, developed in several European research projects. It is based on concepts from UML, SysML and AADL, but adapted for automotive needs and compliance with AUTOSAR. EAST-ADL has been designed to complement AUTOSAR with descriptions at higher level of abstractions: vehicle features, functions, requirements, variability, software components, hardware components and communication

EAST-ADL primary focus is the development of safety-related embedded control systems. It supports the main phases of software development, from early analysis via functional design to the implementation and back to integration and

validation on vehicle level. The main role of EAST-ADL is that of providing an integrated system model. Its focus is on

- documentation, as a single integrated model of a system;
- communication between engineers, through synthetic views on the system;
- analysis, through the description of system structure and associated non-functional properties. This view focuses on an analytic decomposition of a system, not on the analysis of its properties per se;
- simulation or code generation, through behavioral models such as a subsystem in MATLAB/Simulink [9].

In [10] and [11], authors discuss analysis of EAST-ADL models, focusing on model checking using SPIN, safety analysis using Hip-Hops and some timing analysis. This work is later completed in [12] and addresses optimizations of architectures through specific analysis combined to multi-domain optimization techniques based on genetic algorithms. In these experiments, analysis is through exogenous transformations.

C. AADLv2

The "Architecture Analysis and Design Language" (AADL) [6] is a textual and graphical language for model-based engineering of embedded real-time systems. AADL is used to design and analyze software and hardware architectures of embedded real-time systems.

The AADL allows for the description of both software and hardware parts of a system. It focuses on the definition of clear block interfaces, and separates the implementations from these interfaces. From the separate description of these blocks, one can build an assembly of blocks that represent the full system. To take into account the multiple ways to connect components, the AADL defines different connection patterns: subcomponent, connection, binding.

An AADL model can incorporate non-architectural elements: non-functional properties (execution time, memory footprint, ...), behavioral or fault descriptions. Hence it is possible to use AADL as a backbone to describe all the aspects of a system. Let us review these elements:

An AADL description is made of *components*. Each component category describes well-identified elements of the actual architecture, using the same vocabulary of system or software engineering. The AADL standard defines software components (`data`, `thread`, `thread group`, `subprogram`, `process`) and execution platform components (`memory`, `bus`, `processor`, `device`, `virtual processor`, `virtual bus`) and hybrid components (`system`) or imprecise (`abstract`).

Component declarations have to be instantiated into subcomponents of other components in order to model an architecture. At the top-level, a system contains all the component instances. Most components can have subcomponents, so that an AADL description is hierarchical. A complete AADL description must provide a top-most level system that will contain certain kind of components (*processor*, *process*, *bus*, *device*, *abstract* and *memory*), thus providing the root of the architecture tree. The architecture in itself is the instantiation of this system, which is called the *root system*.

The interface of a component is called *component type*. It provides *features* (e.g. communication ports). Components communicate one with another by *connecting* their *features*. To a given component type correspond zero or several implementations. Each of them describes the internal structure of the components: subcomponents, connections between those subcomponents. They can also refine non-functional properties.

The AADL defines the notion of *properties*. They model non-functional properties that can be attached to model elements (components, connections, features, instances, etc.). Properties are typed attributes that specify constraints or characteristics that apply to the elements of the architecture such as clock frequency of a processor, execution time of a thread, bandwidth of a bus. Some standard properties are defined, e.g. for timing aspects; but it is possible to define new properties for different analysis (e.g. to define particular security policies). Besides, the language is defined by a companion standard document that defines legality rules for component assemblies, its static and execution semantics.

AADL initial requirement document mentions analysis as the key, objective. AADL is backed with a large set of analysis tools[2], covering many different domains: scheduling analysis like Cheddar [13] and MAST [14]; dependability assessment: AADL provides an annex for modeling propagation of error, like COMPASS project [15], or ADAPT [16]; behavioral analysis: mapping to formal methods and associated model checkers have been defined for Petri Nets [17], RT-Maude [18] and many others code generation: Ocarina implements Ada and C code generators for distributed systems [19]; mapping to hardware description language System-C [20].

D. Comparison

Actually, all three modeling frameworks stand equal: gateways can be established between the notations. In [21], authors compare EAST-ADL and AADL, and conclude that they have lot of similarities, yet EAST-ADL primary focus is on model exchange and understandability, while AADL focuses more on analysis and code generation. In [22], authors demonstrate that AADL concepts can be expressed using MARTE; another group explored similar considerations from SysML [23].

We note most of the differences are not in the notation itself, but on the way it is used, and its capability to be extended:

- *Coverage of the V-cycle*: MARTE/SysML combined focuses on the whole engineering cycle, whereas EAST-ADL and AADL focus more on the architectural definition, down to code generation and V&V activities.
- *Extensibility*: all three notations allow for extensions through model-based specific constructs: stereotypes, specific NFPs. The preservation of the semantic, and consistency in case of heterogeneous extensions, e.g. for reliability, and targeting (implicitly) different tools;

[2]An updated list of supporting tools, projects and papers can be found on the official AADL web site http://www.aadl.info.

Another aspect is on the coverage of analysis provided. Without much surprise, we not the three notations propose very similar support for schedulability or reliability analysis, but also code generation. Main differences stem in the tools or methods used, not in the information found in the models.

Yet, we note similar shortcomings with respect to analysis:

- Analysis concepts (non-functional properties, topologies, etc.) are part of the modeling space; yet the analysis itself is external, done through an exogenous model transformation targeting another tool. Adding a new simple analysis cannot be done by the designer, but requires tool expertise.
- Addition of new concerns is done through the refinement or addition of new properties, but also through specific modeling patterns. We claim this could jeopardize the genericity of the modeling framework itself. One risk is for instance to use two sets of NFP for related concerns such as safety and reliability. This could later introduce model discrepancies.
- Results of the analysis are not preserved: analysis is viewed as an additional semantic check to confirm the model is correct with respect to particular verification objectives, even-though the associated computation time can be high. This is a short-term view; the result of an analysis can be later used to refine some system metrics. Furthermore, in case of model modification, it is unclear whether an analysis should be redone or not.

These shortcomings greatly reduce the usability of models in the building of large-scale systems. Analysis is reduced to a model validation that is performed depending on the maturity of the model. We claim analysis concerns should be an integrated part of the modeling process, and be expanded beyond typical modeling patterns and NFP. We review several usage scenarios in the next section.

III. Analysis as a first-class citizen: why?

Through the manipulation of models, designers and architects master concepts and technologies involved in the system they deal with, being able to deliver a well-structured product intended to a given purpose. In complement to the modeling activities, analyses applied directly or indirectly onto the models are helpful to assess the quality of the delivered product.

In the previous section, we reviewed three key standardized languages for the modeling of embedded systems, their similarities, and how they address analysis as an objective. In this section, we outline some important issues and challenges pointing the fateful role of analyses in the design process. We motivate and illustrate the challenges with use cases encountered in the literature and provided solutions.

A. Enforcing models consistency

In the case of design space extension, such as the addition of new components, properties or connections, it is not impossible to introduce hidden errors and cause undesired problems; overwriting of previously defined properties, data or interfaces inconsistencies are some examples. The fact is the designer lacks of support to fully control and master the design space.

Actually, the modeling framework should perform background consistency checks (which are sort of analyses) to ensure that a) the modeling artifacts are correctly used and b) evolutions do not break the initial meta-model assumptions. These are *constraints* applied to models: each rule enforces particular restrictions in the way model entities are built or combined. Models can be constrained through specific rules applied directly on the initial meta-model, or through external constraints applied on the meta-model or model instances. As an example, the Object Constraint Language (OCL) can be used to enforce that strategy on UML diagrams [24].

Second question is to know when to attach a constraint to a model or a model element. For instance, for large systems, it is likely that two subsystems will be built on top of incompatible restrictions. For instance, an Unmanned Aerial Vehicle (UAV) ground station can relax many of the restrictions to be applied on the UAV itself. Therefore, one needs a convenient way to weave constraints and model entities.

"REAL" (Requirement Enforcement and Analysis Language) [25] is an annex language that applies on AADL models. It provides a basic mathematical notation, bound to AADL concepts, to express static invariants a model must enforce. This language has been deployed in various settings to ensure a model conforms to architectural patterns (e.g., ARINC653, MILS, Ravenscar) but can also serve extensions and ensure a model conforms to assumptions of a given computational model. Such approach proved to be convenient: one can select granularity of restrictions to be applied on model elements, or the whole.

One other approach is to define contracts attached to model elements so as to formalize the composition of model entities [26]. Yet, the scope is reduced to component interfaces.

B. Facing analysis complexity

We note analyses are truly diverse in complexity, and that the *value* attached to one result is not correlated to its implementation *complexity*.

As an example, in [27], authors performed an architectural analysis of a satellite mission, linking performance metrics to analysis objective such as weight, energy or mission requirements (e.g. number of devices required). Surprisingly, many of those analyses are basic computations of the architectural graph (e.g. summing the weight of each element of the designed system to check the overall weight of the system).

Such simple operations can be implemented with a constraint language like OCL or REAL operating directly on the design model.

Yet, other analyses are much more complex. For instance, let us consider the design of a DRE system subject to real-time or safety constraints. Such a design requires to apply appropriate analyses including schedulability analysis or safety analysis that may involve analytical methods [13] or model checking applied considering a scenario expressed in temporal logic [17]. It is obvious that the case of those "advanced"

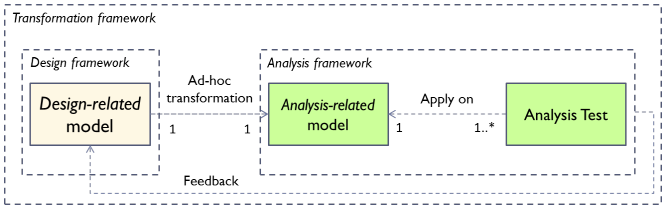

Fig. 1. A classic analysis strategy : the one-to-one transformation workflow

analyses requires more specific analysis methods performed by standalone analysis engines. Consequently, filling the gap between modeling and analysis calls for dedicated support in order to navigate between design model(s) and analysis method(s).

Up to now, this problem is handled in a one-to-one transformation fashion (Figure 1). Mostly (if not always), considered analyses are hard-wired in a transformation framework that will translate the *design-related* model to a "proprietary" *analysis-related* model responding to the execution needs of a specific analysis engine.

C. Handling models evolution

In line with the previous concerns, we note the question of model maturity is central in the engineering of embedded systems. The key question is to know *when* to move from one modeling activity to one analysis phase.

First of all, let us note that each analysis comes with its set of expectations; for instance, in terms of presence of NFPs, or specific communication patterns [28].

Gaudel et al. [29] addressed the interoperability between design models and analysis tools, introducing the notion of *subset*. They specified them as specific restrictions (so-called cardinality constraints) over the design-related meta-model and implemented them using a Domain-Specific Language (DSL).

In [30], the authors introduced the concept of *real-time context* as being attached to a real-time system model. It is defined as a specific set of assumptions on the *design-related* model, compliant to a precise *analytical* model onto which an analysis can be performed (task model, hypothesis, etc.). Given the real-time context of a model, it is possible to execute associated analyses. In this work, detecting real-time contexts is enforced through structural rules applied onto the model instances using OCL.

In both cases, the notion of constraint (as introduced in the section III-A) has been extended to implement a set of *predicates* on the design model. These are used so as to know whether a given analysis can be triggered prior to run the model transformation (see figure 1).

On the one-hand, predicates act as a validation engine prior to perform the actual analysis. On the other hand, from the error report they produce, they act as "wizards" that point to model missing bits, thus providing methodological support to the designer.

D. Exploiting analyses outcomes

Designers can gain more information on their models through analysis. Yet, analysis is not a dead-end: from an analysis outcome, the designer can validate, correct or refine his system; or extend it adding another view, for instance in the MARTE or AUTOSAR. So far, the one-to-one transformation fashion (as mentioned in the section III-B), hardly eases *feedbacks* of analysis results in the design space (see figure 1). Because models and analyses are loosely-coupled, the semantics could be changed during the transformation process, implying that computed results have a weak meaning *vis-a-vis* the initial models. For the best of our knowledge, inclusion of analysis outcomes in the context of models correction or refinement has been relatively unexplored so far. Analysis results can be considered in two ways:

1) Validation capabilities: in case both the input(s) and the output(s) of an analysis are part of the model, applying an analysis can ensure the model elements are coherently defined regarding each other. This gathers two situations:

- model over-engineering : model elements can be *a priori* deduced from others,
- model inconsistency : refers to the *a posteriori* detection of mismatch between modeling elements; a mismatch being the sign of a bogus design.

The case of over-engineering has been investigated in [31] : the authors showed it is possible to deduce configuration parameters for an embedded network using non-functional properties of data flows and threads present in an AADL model.

2) Analysis factory: in case the output(s) of an analysis can feed another analysis, one can derive wider results by reviewing the "interfaces" associated to the analyses. In this case, any analysis can be seen as a black box that implements a transformation function (the analysis itself) regarding *required* inputs (the parameters to analyze) and *provided* outputs (the computed result(s)).

Such "analysis factory" has been experimented in the engineering of the Perseus rocket [32]. This sub-sonic launcher has to meet particular performance metrics to ensure launch success. AADL and REAL languages have been used to respectively define (a) the system components and their NFPs and (b) the computation of these metrics. The computation of all metrics is actually a tree, mixing data from the model and intermediate computations. Using the signature associated to each computation, REAL orders the successive computations and triggers the necessary analyses. Let us note that some analyses are simple computations while others may rely on external tools.

E. Summary : addressing analysis intends

Early resources provisioning, design space exploration, multi-criteria design optimization [33], [12] or, more classically, final validation of a complete system are some examples of *analysis intends*. Actually, such intends are different levels for a same philosophy : models, as time-variant artifacts, are

dynamically assessed and checked on a multi-criteria basis with intervention of the user or not.

Addressing analysis intends calls for a clear and well-structured coupling between model(s) and applicable analyses, thus requiring answering three questions:

1) Is the model well-formed and, Is there any analysis applicable to the *current* model? (Sections III-A, III-C)
2) How will the analysis be performed? (Section III-B)
3) What can we learn from the analysis? (Section III-D)

From the experiments outlined for each challenge, we note a convergence between modeling and analysis concerns:

- *constraints* in the form of restrictions to be applied on the model under construction are mandatory so as to 1) avoid ill-formed models and 2) respect context-specific requirements (e.g. from specific models of computation, or modeling guidelines);
- some "simple" analyses can be performed using a basic DSL, still contributing significantly to the assessment of the system quality. Other analyses require a transformation automaton to translate the design model into a third-party model exploited by the analysis engine;
- third-party *computations* are based on information extracted from the design model(s). Constraints can be used to restrict the scope of computations (e.g. detecting NFPs or patterns belonging to a specific analytical model), thus maintaining the semantics of the model transformation;
- exploiting analysis results in the design space requires a systematic and formalized approach so as to 1) structure and exploit the relationships between analyses and then 2) validate the design choices.

Hence, *modeling* and *analysis* are dual activities: models are built with an analysis objective, while analysis can provide guidance for the model(s) definition. Besides, dealing with the numerous couplings between the two activities – referred to as under multiple names such as "constraints", "restrictions" or "contracts" – calls for a more systematic approach for the use of analyses.

IV. Analysis as a first class citizen: how?

As we mentioned, designers have multiple analysis objectives. Addressing such analysis intends requires embedding solutions within the model-based environment at very various levels (that is to say, to reason at meta or instance levels for instance). As a matter of fact, we note that any analysis activation is made through the use of static predicates whose evaluation to false means the impossibility to proceed further.

As part of the AADL working group, a proposal emerged to merge various initiatives: ACL – AADL Constraint Language – and its roadmap.

A. ACL – An AADL annex language for analysis

The analysis objectives of a model can cover heterogeneous concerns like scheduling, security, safety, power consumption ... Besides, the evaluation of metrics can be project or platform dependent, and rely on different analysis frameworks.

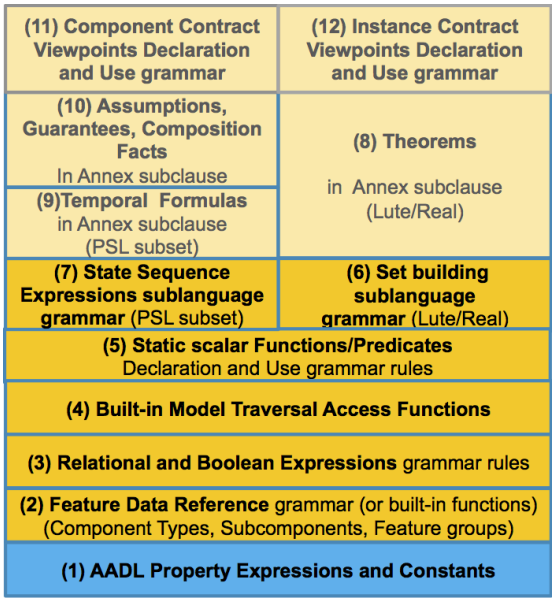

Fig. 2. Layers of ACL – AADL Constraint Language

Thus, one needs a versatile way to define and combine analyses.

These considerations led to the definition of ACL – AADL Constraint Language. Its is defined as an AADL annex language and is resulting from various experiments made in the scope of languages such as REAL (Requirement Enforcement Analysis Language) [25] or Lute [34] that share similarities in terms of philosophy and syntax.

This language aims at defining analysis strategies on architectural descriptions. ACL 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 the UML Object Constraint Language (OCL) and decided to define a specific DSL based on AADL concepts to ease writing of analysis.
- Allowing to define generic rules. We note that mathematics universal quantifiers ($\forall$, $\exists$) notation is interesting to define metrics that can apply to a wide range of models, not just specific instances.
- Allowing for modularity through definition of separate analyses that can be later combined.
- Being integrated to the AADL as an annex language, so that analysis are coupled to models in a single repository.

B. Definition of ACL

ACL is built on top of a family of languages (figure 2); each language aims at modularizing accessors on model elements, definition of analysis predicate for both static and dynamic cases and finally coupling of such predicates to the model.

1) *Model accessors*: extraction of NFPs (1); access to component interfaces (2); model traversal functions (4)
2) *Basic computations*: Relational and Boolean expressions grammar rules (3), Reusable static scalar functions /

predicates (5), Set building operators and reusable set building functions grammar (6 & 7)

3) *Dynamic predicates* based on state sequence expressions from PSL SERE subset (7); subset of PSL Foundation Logic (LTL) temporal operators on Boolean and State Sequence Expressions (9)
4) *Static predicates* based on Lute/Real languages (8)
5) *Modularity* Grouping predicates into standard or custom defined analysis viewpoints (10, 11 & 12)

From these goals, we defined ACL with the following design decisions: the language is based on set theory and associated mathematical notations. The basic unit of ACL is a theorem. A theorem verifies an expression over all the elements of a set that is called the range set. It allows one to build sets whose elements are AADL entities (connections, components or subprogram calls). Verification or computations can then be performed on either a set or its elements by stating Boolean expressions. Listing 1 illustrate some elements of ACL syntax from the PALS case study from [34].

```
-- A structural contract is a set of static predicates
-- a model must meet to be correct
structural contract PalsChecks {
theorem PALS_Period
  foreach s in PALS_Threads do {
    -- Call to a function
    PALS_Group := PALS_Group(s);
    Clock_Jitter := Max_Thread_Jitter(PALS_Group);

    -- Accessing the maximum value of a property on a
    -- set of model elements

    Max_Latency := Max({Upper(Property(c, "Latency"))
      for c in Connections_Among(PALS_Group)});
    Deadline := Property(s, "Deadline");
    PALS_Period := PALS_Period(s);

    -- Structural assertion to be verified
    check (Deadline
       < PALS_Period - 2 * Clock_Jitter - Max_Latency);
  };
end PALS_Period;

-- Grouping of atomic analysis
PALS_Requirements_Verification:
      check theorem PALS_Period_is_Period;
      check theorem PALS_Group_shares_PALS_Period ;
      check theorem PALS_Causality;
      check theorem PALS_Period;
};
```

Listing 1. ACL example

From this set of definitions, one may now attach constraints to model elements using the annex subclause mechanism of AADL. This allows one to attach extraneous information to model entities.

```
system Pals_System_Example
   -- some features declaration
annex Constraint_Annex {**
  -- Any implementation of Pals_system_example must
  -- respect the following contract
   enforce structural contract PalsChecks;
**}
end Pals_System_Example;
```

Listing 2. Attaching constraints to model elements

Using AADL extensions and inheritance mechanism, one can later propagate predicates to component implementation, subcomponents or extensions, thus controlling the scope of a set of analysis. This mechanism allows for a fine-grain control of the constraints to be applied on a system.

C. Status and roadmap

AADL Constraints Annex relies on many previous experiments around AADL in the definition of a language for analysis models: REAL and Lute. Annex main goal is to turn those experiments into a concrete standard to reduce the gap between model and V&V activities.

A first prototype, derived from REAL is under implementation. The core of REAL is preserved by ACL; the main modification concerns the grouping of predicates into analysis contracts. We are in the process of porting all existing library of analysis we did in the past for AADL subsets: IMA, ARINC653, Ravenscar, Time-Triggered, Synchronous and project-specific analysis: PERSEUS rocket, UAVs.

Future work will consider the extension to dynamic predicates, using concepts from the IEEE PSL Language for specifying expected observable behaviors in the system.

As defined, ACL has clear interfaces towards AADL elements:properties, model traversal functions and constraints binding to model elements. A similar approach could be made available to MARTE or AUTOSAR, allowing for similar analysis to be run on heterogeneous models in a designer-friendly mode.

V. Conclusion

The advance of Architecture Description Languages in the scope of Model-Driven Engineering provides foundation for system analysis through model transformation and exploration. Numerous contributions made proposals to bring verification and validation capabilities to designers through model-based frameworks and illustrate benefits to the overall system quality.

Reviewing the major standards for the modeling of DRE systems, we noted that model-level analyses are usually performed as an exogenous, unidirectional, transformation to a third-party model. We notes this narrow the scope of analysis to a one-step process that belongs to tool experts.

We provided arguments to support the idea that analysis should be first-class citizen, at model level. Actually, many analyses are concerned with the correctness of the models, and are bound to domain-specific extensions, enforcement of particular modeling guidelines. In the later case, such guidelines are either process-specific, or encode assumptions done by an analysis plug-in (e.g. for schedulability or reliability).

To help in the definition of such analysis, we first note those are reducible to a set of static or dynamic constraints to be applied on model entities. Such constraints based on model queries allow for validating particular combination of model elements match expected results.

We then introduced ACL, AADL Constraint Languages, as an ongoing work in implementing support for such predicates. ACL relies on previous experiment that demonstrated the versatility of such approach to support various kind of analysis.

Future work will consider the extensions of ACL towards dynamic predicates, and the experimentation of wider test case for Perseus launchers and UAVs designed at ISAE.

REFERENCES

[1] D. Redman, D. Ward, J. Chilenski, and G. Pollari, "Virtual Integration for Improved System Design," in *Proceedings of The First Analytic Virtual Integration of Cyber-Physical Systems (AVICPS) Workshop*, (San Diego, California, USA), November 2010.

[2] H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Maté, K. Nishikawa, and T. Scharnhorst, "Automotive open system architecture-an industry-wide initiative to manage the complexity of emerging automotive e/e-architectures," *Convergence*, pp. 325–332, 2004.

[3] P. Cuenot, P. Frey, R. Johansson, H. Lonn, Y. Papadopoulos, M.-O. Reiser, A. Sandberg, D. Servat, R. Tavakoli Kolagari, M. Torngren, and M. Weber, "The EAST-ADL Architecture Description Language for Automotive Embedded Software," in *Model-Based Engineering of Embedded Real-Time Systems*, pp. 297–307, Springer Berlin Heidelberg, 2011.

[4] B. Selic and S. Gerard, *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*. The MK/OMG Press, Elsevier Science, 2013.

[5] T. Weilkiens, *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[6] SAE, "Architecture Analysis and Design Language (AADL) AS-5506A," tech. rep., The Engineering Society For Advancing Mobility Land Sea Air and Space, Aerospace Information Report, Version 2.0, January 2009.

[7] C. Mraidha, S. Tucci-Piergiovanni, and S. Gerard, "Optimum: A MARTE-based Methodology for Schedulability Analysis at Early Design Stages," *SIGSOFT Softw. Eng. Notes*, vol. 36, pp. 1–8, Jan. 2011.

[8] S. Bernardi, J. Merseguer, and D. Petriu, "A dependability profile within MARTE," *Software & Systems Modeling*, vol. 10, no. 3, pp. 313–336, 2011.

[9] A. Tewari, *Modern control design with MATLAB and SIMULINK*. Wiley Chichester, 2002.

[10] D. Chen, L. Feng, T.-N. Qureshi, H. Lonn, and F. Hagl, "An architectural approach to the analysis, verification and validation of software intensive embedded systems," *Computing*, vol. 95, no. 8, pp. 649–688, 2013.

[11] D.-J. Chen, R. Johansson, H. Lönn, H. Blom, M. Walker, Y. Papadopoulos, S. Torchiaro, F. Tagliabo, and A. Sandberg, "Integrated safety and architecture modeling for automotive embedded systems," *Elektrotechnik und Informationstechnik*, vol. 128, no. 6, pp. 196–202, 2011.

[12] M. Walker, M.-O. Reiser, S. T. Piergiovanni, Y. Papadopoulos, H. Lönn, C. Mraidha, D. Parker, D.-J. Chen, and D. Servat, "Automatic optimisation of system architectures using EAST-ADL," *Journal of Systems and Software*, vol. 86, no. 10, pp. 2467–2487, 2013.

[13] F. Singhoff, A. Plantec, P. Dissaux, and J. Legrand, "Investigating the usability of real-time scheduling theory with the Cheddar project.," *Journal of Real-Time Systems, Springer Verlag*, vol. 43, pp. 259–295, November 2009.

[14] M. González Harbour, J. Gutiérrez García, J. Palencia Gutiérrez, and J. Drake Moyano, "MAST: Modeling and Analysis Suite for Real Time Applications," in *13th Euromicro Conference on Real-Time Systems*, pp. 125–134, IEEE, 2001.

[15] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri, "The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems," in *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security*, SAFECOMP '09, (Berlin, Heidelberg), pp. 173–186, Springer-Verlag, 2009.

[16] M. Hecht, A. Lam, and C. Vogl, "A Tool Set for Integrated Software and Hardware Dependability Analysis Using the Architecture Analysis and Design Language (AADL) and Error Model Annex," in *ICECCS* (I. Perseil, K. Breitman, and R. Sterritt, eds.), pp. 361–366, IEEE Computer Society, 2011.

[17] X. Renault, F. Kordon, and J. Hugues, "Adapting models to model checkers, a case study : Analysing AADL using Time or Colored Petri Nets," in *IEEE/IFIP 20th International Sypmosium on Rapid System Prototyping* , (Paris, France), June 2009.

[18] P. C. Ölveczky, A. Boronat, and J. Meseguer, "Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude," in *FMOODS/FORTE* (J. Hatcliff and E. Zucca, eds.), vol. 6117 of *Lecture Notes in Computer Science*, pp. 47–62, Springer, 2010.

[19] 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*, vol. LNCS, (Brest, France), pp. 237–250, June 2009.

[20] R. Varona-Gomez and E. Villar, "AADL Simulation and Performance Analysis in SystemC," in *Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '09, (Washington, DC, USA), pp. 323–328, IEEE Computer Society, 2009.

[21] A. Johnsen and K. Lundqvist, "Developing Dependable Software-Intensive Systems: AADL vs. EAST-ADL," in *Ada-Europe 2011* (A. Romanovsky and T. Vardanega, eds.), pp. 103–117, Springer-Verlag, June 2011.

[22] M. Faugere, T. Bourbeau, R. de Simone, and S. Gerard, "MARTE: Also an UML Profile for Modeling AADL Applications," *Engineering of Complex Computer Systems, IEEE International Conference on*, vol. 0, pp. 359–364, 2007.

[23] R. Behjati, T. Yue, S. Nejati, L. Briand, and B. Selic, "Extending sysML with AADL Concepts for Comprehensive System Architecture Modeling," in *Proceedings of the 7th European Conference on Modelling Foundations and Applications*, ECMFA'11, (Berlin, Heidelberg), pp. 236–252, Springer-Verlag, 2011.

[24] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2 ed., 2003.

[25] O. Gilles and J. Hugues, "Expressing and enforcing user-defined constraints of AADL models," in *Proceedings of the 5th UML& AADL Workshop (UML&AADL 2010)*, (University of Oxford, UK), pp. 337–342, Mar. 2010.

[26] A. L. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems," *Eur. J. Control*, vol. 18, no. 3, pp. 217–238, 2012.

[27] J. Delange and J. Hugues, "Incremental modeling and validation of space mission using AADLv2," in *SAE 2011 AeroTech Congress & Exhibition*, 2011.

[28] A. Plantec, F. Singhoff, P. Dissaux, and J. Legrand, "Enforcing applicability of real-time scheduling theory feasibility tests with the use of design-patterns," in *Leveraging Applications of Formal Methods, Verification, and Validation*, pp. 4–17, Springer, 2010.

[29] V. Gaudel, A. Plantec, F. Singhoff, J. Hugues, P. Dissaux, and J. Legrand, "Enforcing Software Engineering Tools Interoperability: An Example with AADL Subsets," in *IEEE International Symposium on Rapid System Prototyping*, (Montreal, Canada), October 2013.

[30] Y. Ouhammou, E. Grolleau, M. Richard, and P. Richard, "From Model-based Design to Real-Time Analysis," in *VALID 2012, The Fourth International Conference on Advances in System Testing and Validation Lifecycle*, pp. 45–50, 2012.

[31] G. Brau, J. Hugues, and N. Navet, "Refinement of AADL models using early-stage analysis methods – An avionics example," Tech. Rep. TR-LASSY-13-06, Laboratory for Advanced Software Systems, 2013.

[32] G. Duval, "Modeling of a supersonic rocket using Architecture Analysis Language," in *Proceedings of the International Astronautical Congress'12*, no. IAC-12,E2,1,5,x16319, (Naples, Italy), oct 2012.

[33] M.-Y. Nam, K. Kang, R. Pellizzoni, K.-J. Park, J.-E. Kim, and L. Sha, "Modeling towards incremental early analyzability of networked avionics systems using virtual integration," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 11, no. 4, p. 81, 2012.

[34] D. D. Cofer, A. Gacek, S. P. Miller, M. W. Whalen, B. LaValley, and L. Sha, "Compositional Verification of Architectural Models," in *NASA Formal Methods* (A. Goodloe and S. Person, eds.), vol. 7226 of *Lecture Notes in Computer Science*, pp. 126–140, Springer, 2012.