Mechanization of a Large DSML: An Experiment with AADL and Coq

Jérôme Hugues, Lutz Wrage
Software Engineering Institute
Carnegie Mellon University,
Pittsburg, PA, USA
{jjhugues, lwrage}@sei.cmu.edu

John Hatcliff
Department of Computing and Information
Kansas State University, Manhattan, KS, USA
hatcliff@ksu.edu

Danielle Stewart
Galois, Inc.
Minneapolis, MN, USA
danielle@galois.com

***Abstract*—Domain-Specific Modeling Languages (DSMLs) rely on model-based techniques to deliver tailored languages to meet specific needs, such as system modeling, formal verification, and code generation. A DSML has specific static and dynamic behavior rules that must be properly assessed before processing the model. The definition of these rules remains a challenge. Meta-modeling techniques usually lack the foundational elements required to fully express behavioral semantics. In this context, using an interactive theorem prover provides a mathematical foundation with which the semantics of a DSML can be defined. This includes an abstract syntax tree, typing rules, and derivation of an executable simulator.**

In this paper, we report on an ongoing effort to capture the SAE AADL language using Coq along with specific analysis capabilities. Our contribution provides an unambiguous semantics for a large set of the language and can be used as a foundation to build rich analysis capabilities.

I. Introduction

The scope of modeling activities has broadened to all domains for which software provides an advantage. In this context, Domain-Specific Modeling Languages (DSML) emerged as a solution to provide domain-specific concepts and notations. They encompass domain-specific languages [1] used to document an existing system or support code generation and the verification of a system.

A DSML may offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application. Although DSL development is challenging and requires both domain knowledge and language development expertise, frameworks exist to reduce the entry barrier. For instance, the Eclipse community proposes Xtext/Xtend [2] to implement textual DSML and the use of Sirius for graphical DSML [3]. But at the same time, this steady creation of DSMLs is echoeing the large number of programming languages that existed in the late 70s, during which the DoD acknowledged that more than 450 languages were in use [4].

Copyright 2022 IEEE. This material is based upon work supported by the U.S. Army Combat Capabilities Development Command, Aviation and Missile Center, under Contract No.W911W6-20-C-2020. Copyright 2022 IEEE. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. DM22-0854.

Defining and implementing a DSML is another challenge. First, the languages must provide foundations for expressing a well-defined set of concepts. Second, they must provide a clear definition of their semantics that is amenable to tool processing. The definition of the static semantics of a DMSL is well-supported through checks executed by visiting the abstract syntax tree of a model. The definition of the dynamic semantics is either ad hoc and encoded in the tool environment itself or made explicit using another notation, e.g. to inject concurrency into a DMSL [5].

In parallel to the definition of DSML framework from the model-based community, the theorem prover community also developed strategies to define DSLs that are mathematically grounded and support a correctness-by-construction approach. Beyond the well-known CompCert project [6], several extensions have been developed recently to assist in defining a DSL in a theorem prover, for instance a Hardware Description Language in Coq [7] and in Isabelle/HOL [8].

In this paper, the authors are interested in the following question: "How can a modern interactive theorem prover like Coq assist in implementing a large DSML and its verification toolchain?". By large, we mean a complete DSML defined by an industry standard that would encompass static and dynamic semantics. Our contribution focuses on defining and implementing the SAE AADL language, a large DSML designed for modeling safety-critical systems. We provide the mechanization using the Coq theorem prover. The AADL language has rich static and dynamic semantics which allows one to model the architecture of real-time safety-crticial systems and perform performance, safety, or security analyses in addition to code generation. Although multiple tools exist to process AADL models, very few capture the entirety of the language dynamic semantics. The Coq theorem prover has a rich set of features to support the definition and implementation of a DSL. We selected Coq because of its large user base and set of libraries.

In the following, we show that Coq is a viable implementation strategy for defining and implementing a DMSL. Our main contributions are:

- a mechanization of AADL instance model;
- Coq notations or a JSON front-end to import AADL model definitions;

- a helper library to build models and prove static properties of a model;
- a case study that illustrates how to perform schedulability analysis of an AADL model using PROSA [9].

These contributions culminate in this experience report where we describe how we defined and implemented a large DSML using Coq. We present the main design decisions we have taken to build a complete Coq library to manipulate models and perform model verification. We show how to build a tool by exercising Coq's code extraction. Finally, we introduce strategies to define the semantics of a large DSML through multiple refinements. All developed artifacts are available for review.

The paper is organized as follows: in Section II we introduce the key features of the Coq theorem prover to support the definition of a DSL, in Section III we introduce AADL and its core concepts, and in Section IV the high-level design of our mechanization. In Section VI we show how to apply this mechanization to evaluate static properties on an AADL model and perform scheduling analysis. Then, we review related work (Section VII) and provide a conclusion.

II. Coq, A Theorem Prover for Implementing a DSL

Coq [10] is a proof assistant, or Interactive Theorem Prover (ITP). A developer may use the Coq language, Gallina, to write mathematical definitions, executable algorithms, and theorems within an Interactive Development Environment (IDE). Coq has been used to prove non-trivial mathematical theorems and also develop formally certified software and hardware. One interesting feature of many ITPs is the capability to generate certified code (e.g. OCaml or Haskell) from Coq definitions. In this context, "certified" means that there exists a proof script – the certificate – that connects the software produced and the proofs that accompany it. Many references exist to learn more about the Coq ITP such as Pierse et al. [11]. In the following, we introduce a minimal set of notions prior to introducing our Coq development.

Coq relies on Gallina, a functional language that acts as a specification language to describe types, functions, and proofs of some statement. Coq's core follows the rules of the Calculus of Inductive Constructions. The basic types in Coq are either propositions, `Prop`, or sets `Set`. All other types are built on top of these two types. One important aspect of Coq is that `Prop` is the type of logical propositions. It denotes the class of terms representing proofs. `Set` represents typical data types.

In Listing 1, we show some fragments of Coq. First, we define a the `component` inductive type, it list all its elements: identifier, category, etc. This `component` type is recursive: one of its elements is actually a list of `components`. Then, we define the `Unfold` function that recursively build the list of components and subcomponents by iterating over subcomponents.

Coq has a rich standard library that defines regular types such as booleans, integers, lists, and typical results on them. Coq type system support polymorphisms and Java-like interfaces (typeclasses). Because `Prop` and `Set` are both types,

```
Inductive component :=
 | Component : identifier →
   ComponentCategory → (* category *)
   fq_name →           (* classifier *)
   list feature →      (* features *)
   list component →    (* subcomponents *)
   list property_association →
   component
 (* .. *)

Fixpoint Unfold (c : component) : list component :=
  c ::
  (( fix Unfold_subs (ls : list component) :=
    match ls with
    | nil ⇒ nil
    | c :: lc ⇒ Unfold (c) ++Unfold_subs (lc)
    end ) (c→ subcomps)).
```

Listing 1. A Coq inductive type

they can be part of expressions. One of the key aspect of Coq is that a proposition is more powerful than a boolean expression. The type `bool` is computational: one can define functions or perform case analysis. On the other hand, the type `Prop` supports universal quantification over elements, that is typical exists (∃) and for all (∀) quantifiers.

When mechanizing a DSML, we focus on proving specific facts as propositions. Decidable propositions are those propositions for which an algorithm exists for computing either a proof of this proposition or its negation. Since proofs are equivalent to programs, this provides the fundations to extract code from those propositions.

Finally, writing proofs can be a repetitive task. Coq provides a rich library of tactics that encode specific reasoning steps. A typical example is `auto` that will prove basic propositions by applying well-known facts and computations. Coq's set of tactics can be extended by the user using the Ltac language.

We illustrate how we leverage Coq features to mechanize AADL in the next sections.

III. AADL, a DSML for Safety-Critical Systems

A. AADL, a primer

The Architecture Analysis and Design Language (AADL) [12] is both a textual and graphical DSML for modeling embedded real-time systems. AADL is used to design and analyze software and hardware architectures of embedded real-time systems.

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

An AADL model can incorporate non-architectural elements such as non-functional properties (execution time, memory footprint, etc.) and 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 some of these elements.

An AADL description is made of *components*. Each component category describes well-identified elements of the actual architecture, using the same vocabulary as 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 (`abstract`).

In order to model architecture, component declarations have to be instantiated into subcomponents of other components. At the top-level, a system contains all the component instances. In this way, an AADL description is hierarchical. A complete AADL description must provide a top-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 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) with which components can communicate with each other through *connections*. A component type may have zero or more corresponding implementations. Each implementation describes the internal structure of the component such as subcomponents and connections between them. A component type (or implementation) may extend another component so as to refine properties or subcomponents or to add new ones.

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, or the bandwidth of a bus. Some standard properties are defined, e.g. for timing aspects; but it is possible to define new properties to support an analysis.

B. AADL family of semantics

AADL semantics is defined by its standard document. Its definition follows a layered approach. AADL has both static and a dynamic (behavioral) semantics.

1) Static semantics: AADL static semantics is defined through an abstract syntax (a meta-model) along with a set of rules. The figure 1 provides a synthetic view of the AADL meta-model, and shows the relations between all elements. This meta-model is complemented with three set of rules:

- *naming rules* define the notion of namespaces. Namespaces include inside packages and inside components types and implementations. They also defines constraints on the identifier used;
- *legality rules* define rules for composing components into a hierarchy. Legality rules cover connection between components, components than can be subcomponent of others, and property typing.

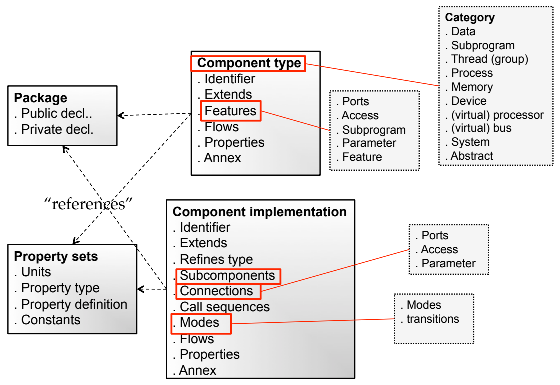

Fig. 1. Simplified view of AADL meta-model

AADL defines the notions of declarative and instance models. A declarative model is the set of packages and property sets that form an AADL project. From a given root component (usally a system), an AADL toolset would build an instance model, a concrete instance of a system, by weaving together the components as a hierarchy of components. During the instantiation steps, all types and actual values for components, properties, and connections are resolved. This is equivalent to building and resolving nodes of an abstract syntax tree in a typical compilation process.

Finally, *consistency rules* check that an instance model is correct, e.g. that some property values are consistent.

An AADL instance model is the input for other form of model processing, e.g. model queries and computations or transformations to another model for analysis. This representation is well-established and used by a large set of tools.

2) Dynamic semantics: AADL dynamic semantics are defined as text rules that define how AADL components are executed at run-time. Each component category is defined by one hybrid automata [13]. In addition, a collection of Runtime Services (RTS) defines how a software realization of a component may interact with ports to send and receive data and events. RTS are configured by AADL properties and define the timing of communication, along with error management policies in case a queue has overflowed or an interaction is invalid, e.g. accessing a port at the wrong instant.

The dynamic semantics of AADL is defined as a collection of text; hence, it is difficult to reason on the dynamic semantics of a model.

IV. High-Level Design of a Mechanization of AADL Semantics

In the previous section, we provided a high-level overview of the AADL language. Our objective is to provide a mechanization of AADL that is a faithful transcription of the standards.

A review of the the AADL standard provides a hierarchical decomposition of the language concepts (see Figure 2).

First, key grammar elements are introduced, then each concept is further refined by defining the legality and naming

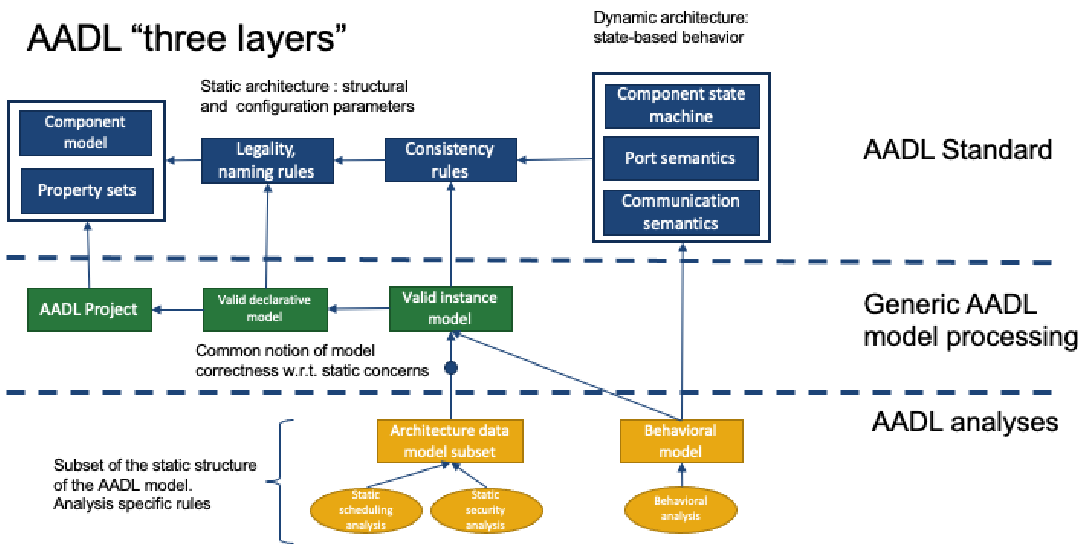
AADL "three layers"
Dynamic architecture: state-based behavior
Static architecture : structural and configuration parameters
Component model
Property sets
Legality, naming rules
Consistency rules
Component state machine
Port semantics
Communication semantics
AADL Standard
AADL Project
Valid declarative model
Valid instance model
Generic AADL model processing
Common notion of model correctness w.r.t. static concerns
AADL analyses
Architecture data model subset
Behavioral model
Subset of the static structure of the AADL model. Analysis specific rules
Static scheduling analysis
Static security analysis
Behavioral analysis

Fig. 2. AADL Standard organization

rules of model elements, consistency rules, and finally their behavior using hybrid automata.

To reduce the gap between the AADL standard specification and our mechanization, we follow a similar approach. First, we provide a generic definition of AADL concepts as a Coq type. Then, we collect each portion of AADL standards related to one pair consisting of a model element and a set of rules, and then we define the corresponding Coq terms and lemmas. Finally, we define the dynamic behavior of AADL model elements. The first step of standard decomposition allows for a recomposition by building either full blown declarative or instance models.

Thus, our mechanization targets the following requirements:

- R1 Our mechanization must be traceable to the original AADL standard document, i.e., every element is a direct translation of a set of definitions from the original document. See Section V-D.
- R2 Our mechanization must support the largest possible subset of the AADL standard. Our objective is to provide a comprehensive mechanization of the standard. From this mechanization, our goal is to allow subsequent development that would leverage this mechanization to implement the analysis. We introduce one such example in Section VI.

Mechanizing an large DSML in a proof assistant entails a large development effort. Therefore, we favor readability over conciseness. In addition, we target the following requirements:

- R3 Our mechanization must provide an easy way to import an AADL model. See Section V-B.
- R4 All rules implemented in our mechanization should be decidable, i.e., one must be able to build a proof that a model either verifies a predicate or does not. This is important to allow for code extraction and the derivation of a tool. See Section V-C.
- R5 Our mechanization should provide utility functions to assess properties over a model and verify them with automation whenever possible. See Section VI;

Ultimately, our mechanization provides a semantics for AADL models. It is axiomatic by essence since it cannot be "proved" to be correct. However, one can validate its correctness by processing specific models. Requirements $R3$ and $R4$ provide the elements to derive a reference interpreter. The results computed by this interpreter are the specification of the meaning of each model.

In the following sections, we introduce incrementally all the elements of our mechanization.

V. AADL Mehcanization – Core Concepts

A. AADL generic component model

At its core, AADL relies on a restricted set of concepts to support its constructs. We informally introduced these concepts in Figure 1. Formally,

Definition 1: An AADL component $Comp$ is a tuple $(c_{id}, cat_C, \mathcal{F}, Prop, \mathcal{S})$ s.t.

c_{id}: the unique component id,

cat_C: the component category,

$\mathcal{F}$: the set of features,

$Prop$: the set of properties associated with this component,

$\mathcal{S}$: the set of subcomponents.

The mechanization of this type in Coq relies on inductive type definitions, i.e. the capability to define a new type from constants and functions that create terms of that type. We illustrate this in the code snippet shown in Listing 2. First, we define `ComponentCategory`, it is an enumeration type that lists all possible component categories. We then provide a joint definition for components, features, and connections. Because these types are mutually dependent, they must be defined in conjunction. The types

DirectionType and FeatureCategory are also enumerations, whereas identifier, fq_name, and feature_ref are variants of string-like type to denote respectively an identifier, a fully qualified name, and a patch (a list of identifiers). They are omitted for conciseness.

```
Inductive ComponentCategory : Type :=
 | system | abstract | process | thread
 (* [..] *)

 Inductive component :=
  | Component : identifier →
    ComponentCategory → (* category *)
    fq_name →           (* classifier *)
    list feature →      (* features *)
    list component →    (* subcomponents *)
    list property_association →
    component
 with feature :=
  | Feature : identifier →
     DirectionType →
     FeatureCategory →
     component →
     list property_association →
     feature
 with connection :=
  | Connection : identifier →
     feature_ref →
     feature_ref →
     connection.
```

Listing 2. A Coq inductive type

The definitions shown in Listing 2 are understood as follows. A connection can be built using the Connection constructor and three parameters: an identifier and two feature references.

Another key aspect of an AADL model is the concept of property association, linking a property value (e.g. 42) to a property type (e.g. Priority) that applies to component of a specific category such as threads. The definition of those types follow the same pattern.

An important concept in Coq is that programs and proofs are equivalent. This applies to all concepts, including equality. Coq relies on so-called Leibniz equality which states that, given any x and y, $x = y$ if and only if, given any predicate P, $P(x)$ if and only if $P(y)$. Building such proof could be seen as a tedious and repetitive effort. Fortunately, Coq provides appropriate tactics to derive these equality princples.

B. AADL concepts as a Coq DSL

Using Coq constructors to build a full AADL model results in cumbersome syntax constructs as shown in Listing 3. A solution is to use Coq notations: advanced commands to modify the way Coq parses and prints objects. A notation is an alternate syntax for entering or displaying a specific term or term pattern and is well-used [7].

A notation is a construct similar to regular expressions that maps elements to arguments of some constructors. This is shown in the code snippet of Listing 4. In this example, we

```
Example A_Component := Component
(Id "a_component")
(abstract)
(FQN [Id "pack1" ] (Id "foo_classifier") None)
nil nil nil nil.
```

Listing 3. Building AADL components

present the notation to parse an abstract component and build the corresponding Coq term along with an example of use.

```
 Notation "'abstract:'
 id →| classifier
 features: lf
 subcomponents: ls
 connections: lc
 properties: lp" :=
 (Build_Component id abstract classifier lf ls lc lp)
 (at level 200).

Example A_Component_2 :=
 abstract: "a_component" →| "pack1::foo_classifier"
  features: [
   feature: in_event "a_feature" ;
   feature: out_event "a_feature2"
  ]
  subcomponents: [
   thread: "a_thread" →| "pack2::thread_t"
    features: nil
    subcomponents: nil
    connections: nil
    properties: nil
  ]
  connections: [
   connection: "c1" *"a_feature" −→ "a_feature_2"
  ]
  properties: nil.
```

Listing 4. Notation for AADL

With this notation, we provide a way for the user to define AADL models. We devised the notations to be close to the AADL original syntax. An alternative strategy not presented in this paper is to import a JSON serilization of an AADL model in Coq and parse it.

C. AADL rules as decidable properties

The previous elements can be used to build AADL models elements that are typed statically by Coq rules. The AADL standard defines additional rules that judge the correctness of an AADL model through naming, legality, and consistency rules (introduced in Section III-B).

These rules, in their most general form, are defined in the standard as natural language predicates. For instance naming rule $4.5(N1)$ states that “The defining identifier of a subcomponent declaration placed in a component implementation must be unique within the local namespace of the component implementation that contains the subcomponent.” In other words, the list of identifiers built from the list of subcomponents has no duplicates.

Such a predicate has a direct translation in Coq: we build the list of the subcomponent identifiers and check there is no duplicate. This rule is obviously decidable: one can derive a proof for either the true or false case by computing the term explicitly.

We implemented most rules defined by the AADL standards. They have a uniform pattern: a predicate that combines conjunctions or disjunctions and the evaluation of properties on lists (test for duplicated elements, inclusion, etc.). This is a direct consequence of the general structure of the AADL and Coq languages. AADL model elements are components, and their well-formedness depends on the way their features, connections, and subcomponents are defined, all of which are built on top of Coq basic types and lists. Hence, the implementation in Coq of these rules has no practical difficulty. Likewise the proof of their decidability.

In addition, we defined a specific induction principle to extend the previous rules to the case of a hierarchy of components. This induction scheme is similar to a visitor pattern that will walk through a model and check that each node is defined according to language rules.

Although this step yields no implementation challenge, it proved to be an interesting step to assess AADL rigor in defining its semantics. First, it confirmed that all rules are defined with sufficient details and can be written as formal propositions. Second, it confirmed that all rules are decidable, which is stronger than boolean-based implementation used in other AADL tools.

D. *Coverage of AADL concepts*

One of our goals is to support an extensive subset of AADL as Coq elements. In this section, we introduced the key elements of our mechanization. Our mechanization allows for the definition of AADL property sets and packages. We implemented Coq types, functions, and lemmas that cover the AADL instance model, and the definition of property sets. We provide a full library to manipulate AADL concepts as either native Coq types or through notation.

We left out the following elements of the AADL language, and provide the rationale for these choices:

- Our mechanization covers all AADL concepts except for flows and modes. Flows are annotations on an AADL architecture to mark a particular data flow for an analysis. We do not consider them given their special usage. Modes are specific configurations of a system, e.g. specific values for property values of activated/deactivated components and connections at specific time. The support of modes is a contending issue in the standard. The authors' analysis is that the current definion of modes is not sufficiently precise to warrant a mechanization effort.
- Our mechanization covers only the instance model and discards the declarative model. An AADL instance model is a fully resolved model and is the output of a compilation-like process from a collection of declarative models. An instance model is the input of all AADL-based toolchains, acting as a fully resolved abstract syntax tree.

The resulting Coq library can be used in two different ways. First, code can be extracted to validate the JSON-serialization of AADL models produced by AADL third-party tools[1]. Code extraction requires all lemmas to be proved. In particular, any functions that return, whether or not a model matches specific predicates, should be decidable. All rules stated in the standard fall into this category, and we could generate a proof-carrying oracle that checks the conformance of an AADL model to the standard.

Another option is to use Coq as an interactive theorem prover so that one can proof more complex lemmas. We discuss this in the next section.

VI. Proving static properties on an AADL model

In the previous sections, we presented the implementation of AADL-mandated rules as decidable predicates. This represents only a special case of properties a user may want to evaluate on an AADL model. In this section, we illustrate how this mechanization can be used to evaluate static properties on an AADL model.

A. *Mapping Resolute to Coq*

AADL has a rich ecosystem of user-provided extensions that support a wide range of concerns, e.g. to support specific architecture style like Future Airborne Capability Environment (FACE) or ARINC653, but also language extensions to model behavioral specifications or error propagations [14]. Resolute [15] provides a DSL to define first-order predicates to be evaluated on an AADL model. Resolute defines both a set of accessors over AADL models along with a language for defining predicates as functions, and predicates using logical connectives.

A natural extension of our mechanization is to implement Resolute accessors and weave them with the Coq language for defining predicates. Hence, this provides a higher-level API to access AADL concepts.

Resolute abstracts AADL concepts as `named_elements`, i.e. a component, a feature, or a connection. We provide a subset of these accessors as an illustration.

- `name(<named_element>)` returns the name of the named element.
- `has_property(<named_element>, <property>)` returns true if the named element has the property.
- `property(<named_element>, <property>` returns the value of the property. A default value must be supplied, and it is returned if the element does not have the property value.
- `property(<property>): value` returns the value of the property constant.
- `parent(<named_element>): named_element` returns the parent of the named element. The parent must exist.

[1]Specific references to tools omitted to comply with blind review restrictions.

Note: Coq is built on top of a pure functional language. We extend parent with an additional parameter which is the root scope to look for a parent.

These accessors provide a higher-level set of functions compared to the core mechanization we introduced in the previous section. They also require some form of dispatching to account for the diversity of a named element. It is worth noting that Coq provides the concept of typeclasses that are akin to interfaces in object-oriented programming language.

We use a Coq typeclass to define a common interface for accessing all these elements uniformly from either components, features, or property associations. This provides a lighter and uniform API in an elegant way that reuses the low-level types and mechanization we introduced previously.

An interesting feature of Coq typeclasses is that all predicates proved on the low-level API are also valid in their embedding in a typeclass. In particular, all the Resolute accessors that test the presence of some subcomponent or property values on a specific named element are also decidable. This property makes it possible to prove elements using this high-level API. We illustrate one such example using the PROSA, a Coq development for formally-proven scheduling analysis.

B. Formally-proven scheduling analysis of AADL models

AADL models capture the architecture of safety-critical real-time systems. Schedulability is one of the many non-functional properties that must be assessed. A system is deemed schedulable if all its tasks meet their deadline. There exists a large set of techniques to assess the schedulability of a system, ranging from simulation and model checking to feasability tests that evaluate a predicate on an abstraction of the set of tasks run by a system [16].

Our goal is to define a mapping from an AADL instance model to a PROSA task model. We proceed in three steps:

- First, we define general well-formedness rules for AADL constructs, as a minimal set of information must be present in the model,
- then, we define PROSA-specific well-formedness rules: PROSA can only analyse a specific class of systems,
- finally, we propose a mapping function.

As a goal, we want to allow the execution of this mapping if and only if the model matches all well-formedness rules.

Mapping an AADL component definition to a PROSA model requires us to align definitions and concepts used by both PROSA and AADL. PROSA defines the type `concrete_task` which is a record that captures a typical task specification, following Burns Scheduling Notation [17]. A task is defined as a tuple $\tau = C, D, P$, where C is the capacity of a task, i.e. its CPU usage in time units, D its deadline, and P its priority.

We first define a general predicate that asserts that an AADL thread component is properly configured, i.e., the corresponding properties are set using Resolute accessors as shown in Listing 5. This works in two steps: first, we check that a component has the right set of properties, then we check that for all threads the right set of properties is set. Both predicates are trivially decidable: they call instances of decidable functions.

```
Definition Thread_Has_Valid_Scheduling_Parameters
 (c : component) :=
   is_thread c ∧
   has_property c Dispatch_Protocol_Name ∧
   has_property c Period_Name ∧
   has_property c Compute_Execution_Time_Name.

Definition System_Has_Valid_Scheduling_Parameters
 (r: component) :=
   All Thread_Has_Valid_Scheduling_Parameters
    (thread_set r).
```

Listing 5. Valid AADL task model

PROSA provides a general version of of the well-known response time analysis called aRTA [9] that has additional constraints that must be assessed: the system is uniprocessor and the system only uses fixed priority. Again, this can be assessed directly through these accessors functions. The next step is to define a mapping from a valid AADL model to a PROSA taskset. In this particular case, there is a direct mapping from AADL properties and PROSA task set configuration parameters, up to a normalization phase.

a) units: AADL is a design language and has a rich set of units to define quantities such as time, memory, etc. PROSA follows a more abstract approach for real-time scheduling and only considers an integer-based notion of time. We map all AADL time values (period, deadline) to a least common time unit and then as an integer;

b) task arrival: The AADL property `Dispatch_Protocol` defines the activation of tasks to be one among a fixed list, e.g., periodic, sporadic, or aperiodic. PROSA defines a more general notion of task arrival: it lists all the occurences of a task job within a time window. We can compute such an arrival prefix from the definition of the dynamic semantics of AADL components.

Hence we have defined an unsafe mapping function from AADL to PROSA. It is unsafe because this function does not check the validity of the model, rather it blindly translates it. Thus, we define a variant that operates only on those models that match aRTA hypotheses: to call this function, one has to prove first that the model is correct.

In the code snippet of Listing 6, we first present the safe mapping function. Assuming `A_system` is a Coq term denoting an AADL model, we first build a proof that this system meets the aRTA hypotheses and then use this proof to invoke the mapping function. Fortunately, the burden of the proof is not an issue thanks to Coq automation. See the example below.

From the execution of the mapping function, one can now use PROSA's set of techniques and tactics to demonstrate that the system is fully schedulable. Our development provides the full proof on an elaborated example. Let us simply note that the proof itself is a direct application of selected strategies.

```
(* Safe mapping from AADL to PROSA *)
Definition Map_AADL_to_PROSA
   (c: {c : component | Check_aRTA_Hypotheses c})
:=
  Map_AADL_to_PROSA_unsafe (proj1_sig c).

Lemma Check_aRTA_Hypothesis_A_System :
  Check_aRTA_Hypotheses A_System.
Proof.
  (* Compute all possible values *)
  compute.

  (* We have several basic conjunctions,
     we split and discharge them *)
  repeat split ; auto.
Qed.

(* Execution of the mapping function *)

Definition ts_aadl :=
  Map_AADL_Root_to_PROSA (exist _
    A_System (* model *)

    (* proof of validity *)
    Check_aRTA_Hypothesis_A_System).
```

Listing 6. Mapping an AADL task model to PROSA

Hence, proving that an AADL model is schedulable using PROSA is done in a fully automated way: from the verification an AADL model complies to PROSA constraints to the execution of the mapping function from AADL to PROSA task sets, and then again its verification.

C. Lessons learned

Having defined a high-level API to access AADL model elements from Coq that mimics existing AADL tools, we have illustrated the feasibility to rethink AADL toolchains. Instead of relying on black-box tools, one can instead use the full power of an interactive theorem prover to define predicates and mapping functions and use them in an interactive way to prove that some static properties hold.

In this example, we relied on PROSA to implement one minimal scheduling analysis tool. As opposed to other strategies, we do not only get a result, but also the proof of a result. This strengthens the confidence in th evaluation of static properties over a model.

VII. Related Works

In this section, we provide a review of related works and how they compare to our contributions. We decompose our analysis of related work in two categories: AADL-related mechanization and general mechanization in the scope of model-based development.

a) Semantics of a subset of the AADL core using Coq: Yang et al. show how to model a synchronous subset of AADL in Coq and provide a full semantics of this particular subset using timed abstract state machines [18]. Hence, this work is restricted to periodic threads and a few communication schemes. In addition, mapping a general AADL model to their mechanization is a complex effort. Our contributions propose two approaches to map AADL to Coq, either through an external JSON import or through AADL notations. Our objective is different is that we aimed at giving a semantics to the complete AADL core standard: we support a larger set of AADL constructs (the AADL instance model) along with simulation capabilities using DEVS.

b) Meta-modeling and Coq: Buisson et al. show how to map an EMF meta-model to Coq inductive types [19]. The CoqTL project [20] extends interactions with EMF to support generic model transformation capabilities. Our work is only partially concerned with model transformation. We concentrate on the validation of models and defined induction principles and tactics to expedite this process.

VIII. Conclusion

The rigorous development of safety-critical systems often uses Domain-Specific modeling Language to capture specific aspects of a system. Many formal behavioral specifications can be used directly. However, higher-level DSMLs such as AADL are also used to capture the full system design. It is therefore important to properly define their semantics in an unambiguous way. An approach is to use an interactive theorem prover like Coq.

In this paper, we presented the main elements of the mechanization of AADL in Coq. Our contribution delivers the mechanization of a significantly large subset of AADL along with verification capabilities. We used multiple features of Coq to define an AST-like structure as a Coq inductive type representing an AADL instance model, along with accessor functions to build decidable properties. We exercised this on the verification of schedulability for some classes of real-time processors on mono-processor systems.

This development has been validated through several examples and tests. From these elements, a designer may use Coq in interactive mode to prove a statement of correctness on the static or behavioral aspects of an AADL model. Furthermore, one can use automated tools to check the correctness of a model through code extraction.

The resulting Coq development represents approximatively $6.5 KSLOCS$ and include examples and testsuite. It demonstrates a first step towards the full mechanization of AADL in Coq. Our development is available as an Open Source software at https://github.com/Oqarina/.

This development opens several possibilies for future work to 1) improve the definition of the standard, 2) provide a reference mechanized semantics, and 3) build proof-carrying tools that extend the confidence of existing tools.

References

[1] M. Mernik, J. Heering, and A. M. Sloane, “When and how to develop domain-specific languages,” *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, dec 2005. [Online]. Available: https://doi.org/10.1145/1118890.1118892

[2] L. Bettini, *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*, 2nd ed. Packt Publishing, 2016.

[3] V. Viyović, M. Maksimović, and B. Perisić, "Sirius: A rapid development of dsm graphical editor," in *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*, 2014, pp. 233–238.

[4] W. A. Whitaker, "Ada—the project: The dod high order language working group," in *The Second ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL-II. New York, NY, USA: Association for Computing Machinery, 1993, pp. 299–331. [Online]. Available: https://doi.org/10.1145/154766.155376

[5] F. Latombe, X. Crégut, B. Combemale, J. Deantoni, and M. Pantel, "Weaving concurrency in executable domain-specific modeling languages," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2015. New York, NY, USA: Association for Computing Machinery, 2015, pp. 125–136. [Online]. Available: https://doi.org/10.1145/2814251.2814261

[6] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, jul 2009. [Online]. Available: https://doi.org/10.1145/1538788.1538814

[7] C. Pit-Claudel and T. Bourgeat, "An experience report on writing usable DSLs in Coq," 2021.

[8] W. Khan, D. Sanán, Z. Hou, and Y. Liu, "On embedding a hardware description language in isabelle/hol," *Des. Autom. Embed. Syst.*, vol. 23, no. 3-4, pp. 123–151, 2019. [Online]. Available: https://doi.org/10.1007/s10617-019-09226-1

[9] S. Bozhko and B. B. Brandenburg, "Abstract response-time analysis: A formal foundation for the busy-window principle," in *32nd euromicro conference on real-time systems, ECRTS 2020, july 7-10, 2020, virtual conference*, ser. LIPIcs, M. Völp, Ed., vol. 165. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 22:1–22:24, tex.bibsource: dblp computer science bibliography, https://dblp.org tex.biburl: https://dblp.org/rec/conf/ecrts/BozhkoB20.bib tex.timestamp: Mon, 21 Dec 2020 13:23:22 +0100. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECRTS.2020.22

[10] Y. Bertot and P. Castéran, *Interactive theorem proving and program development - Coq'Art: The calculus of inductive constructions*, ser. Texts in theoretical computer science. An EATCS series. Springer, 2004. [Online]. Available: https://doi.org/10.1007/978-3-662-07964-5

[11] B. C. Pierce, A. Azevedo de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey, *Logical Foundations*, ser. Software Foundations series, volume 1. Electronic textbook, May 2018.

[12] S. International, *SAE AS5506 Rev. C Architecture Analysis and Design Language (AADL)*. SAE International, 2017.

[13] T. A. Henzinger, "The Theory of Hybrid Automata," in *Verification of Digital and Hybrid Systems*, M. K. Inan and R. P. Kurshan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 265–292.

[14] S. AS2-C, "SAE Architecture Analysis and Design Language (AADL) Annex Volume 1, Revision A," SAE International, Standard AS5506/1A, 2015.

[15] J. Liu, J. D. Backes, D. D. Cofer, and A. Gacek, "From design contracts to component requirements verification," in *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*, ser. Lecture Notes in Computer Science, S. Rayadurgam and O. Tkachuk, Eds., vol. 9690. Springer, 2016, pp. 373–387. [Online]. Available: https://doi.org/10.1007/978-3-319-40648-0_28

[16] F. Singhoff, A. Plantec, P. Dissaux, and J. Legrand, "Investigating the usability of real-time scheduling theory with the cheddar project," *Real Time Syst.*, vol. 43, no. 3, pp. 259–295, 2009. [Online]. Available: https://doi.org/10.1007/s11241-009-9072-y

[17] R. I. Davis, "A review of fixed priority and edf scheduling for hard real-time uniprocessor systems," *SIGBED Rev.*, vol. 11, no. 1, p. 8–19, feb 2014.

[18] Z. Yang, K. Hu, D. Ma, J.-P. Bodeveix, L. Pi, and J.-P. Talpin, "From AADL to Timed Abstract State Machines: A verified model transformation," *Journal of Systems and Software*, vol. 93, pp. 42–68, Jul. 2014. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121214000727

[19] J. Buisson and S. Rehab, "Effective bridging between ecore and coq: Case of a type-checker with proof-carrying code," in *Modelling and Implementation of Complex Systems - Proceedings of the 5th International Symposium, MISC 2018, Laghouat, Algeria, December 16-18, 2018*, ser. Lecture Notes in Networks and Systems, S. Chikhi, A. Amine, A. Chaoui, and D. Saïdouni, Eds., vol. 64. Springer, 2018, pp. 259–273. [Online]. Available: https://doi.org/10.1007/978-3-030-05481-6_20

[20] Z. Cheng, M. Tisi, and R. Douence, "Coqtl: a coq DSL for rule-based model transformation," *Softw. Syst. Model.*, vol. 19, no. 2, pp. 425–439, 2020. [Online]. Available: https://doi.org/10.1007/s10270-019-00765-6