A Formal Analysis Framework for AADL

Stefan Björnander\textsuperscript{1}, Cristina Seceleanu\textsuperscript{2},
Kristina Lundqvist\textsuperscript{2}, and Paul Pettersson\textsuperscript{2}

\textsuperscript{1} CrossControl AB
Kopparlundsvägen 14, 721 30 Västerås, Sweden
stefan.bjornander@crosscontrol.com
\textsuperscript{2} School of Innovation, Design, and Engineering
Mälardalen University, Box 883, 721 23 Västerås, Sweden
\{cristina.seceleanu, kristina.lundqvist, paul.pettersson\}@mdh.se

Abstract. As system failure of mission-critical embedded systems may result in serious consequences, the development process should include verification techniques already at the architectural design stage, in order to provide evidence that the architecture fulfils its requirements. The Architecture Analysis and Design Language (AADL) is a language designed for modeling embedded systems, and its Behavior Annex defines the behavior of the system. However, even though it is an internationally used industry standard, AADL still lacks a formal semantics and is not executable, which limits the possibility to perform formal verification. In this paper, we introduce a formal analysis framework for a subset of AADL and its Behavior Annex, which includes the following: a denotational semantics, its implementation in Standard ML, and a graphical Eclipse-based tool encapsulating the implementation. We also show how to perform model checking of AADL properties defined in the Computation Tree Logic (CTL).

1 Introduction

Mission-critical embedded systems play a vital role in many applications, like air traffic control and aerospace applications. As system failures may result in serious consequences, the development process should include verification techniques, in order to provide evidence that the system’s architecture fulfills its requirements. The architectural design phase is of high practical interest, since architectural mistakes that cause a system to fail certain requirements are hard and expensive to correct in later development phases.

The Architecture Analysis and Design Language (AADL) [1] is a standard of the Society of Automotive Engineers (SAE\textsuperscript{3}), and is based on MetaH [2] and UML [1]. AADL is designed for modeling both the hardware and the software of embedded systems. The standard includes several annexes, out of which the Behavior Annex [3] provides means of describing the behavior of the model.

\textsuperscript{3} SAE is presented at http://www.sae.org.
Even if appealing and already adopted by industry, AADL still lacks a formal semantics, which is particularly important for the design of mission-critical embedded systems, since failures may cause serious damage to people or valuable assets. Such systems are often required to pass certification processes in order to provide sufficient evidence about their safety. Moreover, AADL models are not executable, which limits the possibility to formally analyze their safety and liveness properties.

Consequently, it is highly desirable to overcome such limitations of AADL. To do so, one has to define AADL formally, as any attempt to achieve formal verification requires a precise mathematical method. It is also beneficial that the analysis techniques based on the semantics are supported by tools that are integrated into an AADL tool chain; this would make it easier for an user with limited knowledge of the underlying formalism, to perform, e.g., model checking of AADL models.

In this paper, we introduce a formalization of the meanings of a subset of AADL and its Behavior Annex, in denotational style. Our choice of a denotational style for AADL structures is justified by the simplicity of the semantic models, which is known to improve generality and ease of reasoning [4].

To complete our analysis framework, we also describe the Standard ML implementation that forms the basis of our AADL verification tool, called the ABV tool [5]. The semantics as well as its implementation and CTL verification are illustrated through a small example.

The results of this contribution, together with the recently developed verification tool ABV, define our AADL formal analysis framework, which contains the following:

- A denotational semantics of a subset of AADL and its Behavior Annex, which also includes model checking of properties defined in CTL.
- The semantics implemented in Standard ML, and a parser translating the model defined in AADL and its Behavior Annex, as well as a subset of the CTL property specification, into a format (in Standard ML) suitable as input.
- The ABV model checker [5] that encapsulates the semantic implementation and the parser in a graphical user interface based on the Eclipse Framework. With the help of the tool, the user is able to verify a subset of CTL properties of the model without knowledge of the underlying formalism. The tool has been recently introduced and is out of the scope of this paper.

The rest of this paper is organized as follows. Section 2 overviews preliminaries, describing, among other things, the syntax of the subset of AADL and its Behavior Annex that we focus our work on. Section 3 defines the information gathering part of the denotational semantics. In Section 4, the AADL verification by model checking is described, at a high-level. Finally, in Section 5 we discuss related work, before concluding the paper in Section 6.
2 Preliminaries

In AADL, there are two kinds of systems: the system that defines the port interface and an optional behavior annex, and the system implementation that defines the subcomponents and the port connections between them. In this paper, we have chosen a subset of the AADL model that includes at least one system and exactly one system implementation, which occurs at the end of the definition. The subcomponents of the system implementation are instances of earlier defined systems (equivalent to objects and classes in object-oriented languages), and the connections are made between the input and output ports of the subcomponents, not the systems. The syntax of our AADL subset is given in List. 1.

In order to increase the expressiveness of AADL, it is possible to add annexes. One of them is the Behavior Annex [6,7], which basically models an abstract state machine [8]. Each component of the model describes its logic, by defining a behavior state machine, which consists of the parts State Variables, Initializations, States, and Transitions. The corresponding syntax is given in List. 2.

CTL is a branching-time temporal logic, that is, a language in which time is modeled as a tree structure with a non-determined future. There are several different paths; any one of them may be realized. There are several quantifiers and operators available in CTL: out of them, we will use the universal, all, and existential, exists, quantifiers over paths, together with the global and eventually path-specific operators.

In an AADL model, identifiers are bound to values that need to be stored for further use. Therefore, we need to utilize the data types list, table, set, and tree to store the values (see Björnander et al. [9] for their complete semantic definitions).
3 The Semantics of AADL Structural Elements

In this section, we define the semantics of the subset of AADL and its Behavior Annex described in Section 2. We formalize the meaning of the latter, by constructing mathematical objects, called *denotations* (see functions in Lists 4 and 5). The denotational semantics consists of the mathematical models of meanings (*model* $[S \ S I]$ in List 4 and *system* $[S_1 \ S_2]$ and *system* $[system \ I \ SB \ end ;]$ in List 5), and the corresponding semantics functions, respectively (*model* : Model $\to$ Table in List 4 and *system* : System $\to$ Table in List 5).

In our approach, the semantics can be divided into three phases: information gathering, state space generation, and state space tree evaluation. This section describes the information gathering phase briefly, since it is a rather straightforward process. The other phases are described in more detail in Section 4.

Formally, an AADL system is a tuple $\langle S, s_0, I, Var, P_{in}, P_{out}, T \rangle$, where $S$ is a non-empty finite set of states and $s_0 \in S$ is a compulsory initial state. $Var$ is a, possibly empty, finite set of state variables. $P_{in}$ and $P_{out}$ are the possibly empty finite sets of input and output ports, respectively. $I \subseteq (Var \times Expr) \cup P_{out}$ is a possibly empty set of initializations. $T \subseteq S \times Expr \times S \times A$ is a possibly empty set of transitions, where $A \subseteq (Var \times Expr) \cup P_{out}$ is a possibly empty action set, and $Expr$ can be a state variable, a constant value, or an arithmetic expression. The input port expression is of Boolean type.

The values of a system are formally defined in List 3. As there can only be one system implementation, its subcomponents are stored in the subcomponent list.

In an AADL model, identifiers are bound to values that need to be stored for further use. In order to store these values, several tables and lists are needed:

<table>
<thead>
<tr>
<th>Listing 2 The Behavior Annex syntactic rules.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annex ::= annex Identifier {** StateVariables?</td>
</tr>
<tr>
<td>Initializations? States? Transitions? **} ;</td>
</tr>
<tr>
<td>StateVariables ::= state variables StateVariable+</td>
</tr>
<tr>
<td>StateVariable ::= Identifier : integer ;</td>
</tr>
<tr>
<td>States ::= states State+</td>
</tr>
<tr>
<td>State ::= Identifier : initial state ;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Initializations ::= initializations Action+</td>
</tr>
<tr>
<td>Transitions ::= transitions Transition+</td>
</tr>
<tr>
<td>Transition ::= Identifier - [ Expression ] -&gt; Identifier ;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Action ::= Identifier := Expression ;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Expression ::= Expression ArithmeticOperator Expression</td>
</tr>
<tr>
<td>ArithmeticOperator ::= +</td>
</tr>
</tbody>
</table>
The system table holds the systems of the model. The information of each system is stored in the tuple \(\langle \text{state}, \text{symbol table}, \text{init list}, \text{trans list} \rangle\), where state is the current state of the annex (initialized to zero, representing the initial state), symbol table holds the input and output ports of the system, as well as states and state variables of the annex, init list holds the list of initializations, and trans list holds the list of transitions. For each system, its tuple is associated with the name of the system in the system table.

The subcomponent table and subcomponent list hold the subcomponents of the system implementation. They hold the same subcomponents - the table is used to look up states and state variables in the CTL property specification (see Section 2), and the list is used to keep track of connections between the subcomponents.

The connection list holds the connections between the subcomponents. In order to identify the sending and receiving subcomponents, it uses the index in the subcomponent list above.

The local system table - each system has a symbol table, holding the input and output ports as well as the states and state variables of the behavior annex. Each system also holds a local initialization list and transition list. This information is originally stored for each system and copied to the subcomponents instantiating the systems.

For each syntactic rule of Section 2, one corresponding semantic rule is defined. The semantic rules of this section work in a way similar to a traditional compiler; they gather information that is stored in the structures listed above. Due to space limitation, we confine ourselves to showing the model (List. 4), and system (List. 5) rules, here (see Björnander et al. [9] for the complete definitions of the rules).

All observably distinct elements have distinct denotations in form of semantic functions, which ensures the soundness of the set of semantic rules. The semantic functions are structure-preserving functions, such that each morphism of the
Listing 4 The model semantic function.

\[
\text{model} : \text{Model} \rightarrow \text{Table} \\
\text{model } [S \text{ SI}] = \\
\text{let system_table} = \text{system } S \text{ in} \\
\text{system_impl SI system_table}
\]

Listing 5 The system semantic function.

\[
\text{system} : \text{System} \rightarrow \text{Table} \\
\text{system } [S_1 \text{ S}_2] = \\
\text{let system_table}_1 = \text{system } S_1 \text{ in} \\
\text{let system_table}_2 = \text{system } S_2 \text{ in} \\
\text{table_merge system_table}_1 \text{ system_table}_2 \\
\text{system } [\text{system I SB end }] = \\
\text{table_set I (system_body SB) table_empty}
\]

semantic model is a denotation of an architectural element, which ensures the completeness of the same rule set.

4 Verification by Model Checking

In this section, we describe the verification of CTL properties of AADL models. The main difference between this section and Section 3 is that in Section 3, the semantic rules have been used to gather information about the model, while we, in this section, utilize that information to perform model checking.

The main idea is to generate a state space tree (a state space is the sum of the states of all the annexes of the system, technically, a subcomponent list) that becomes traversed with regard to the CTL property specification.

4.1 Tree Generation

In this section, we generate the state space tree that is initially made of one single node holding the initial state space, that is, the subcomponent list in its initial state. The \text{traverse_subcomponent_list} (List. 7) traverses the subcomponents, and for each subcomponent \text{traverse_transition_list} (List. 8) traverses the transitions. For each transition that can be taken, \text{execute_transition} (List. 9) updates the state space so that the transition is taken, and creates a new sub-tree with the new state space as root value. Then, it attaches the sub-tree as a child tree to the main tree. Finally, it calls \text{generate_tree} (List. 6), which recursively continues to create sub-trees until no more transitions can be taken. However, in order to prevent infinite tree generation, the generation process is aborted if a previous state space reoccurs.
Listing 6 The \textit{generate\_tree} semantic function.

\begin{verbatim}
generate\_tree : List \times List \times Set \times Tree \rightarrow Tree
\end{verbatim}

\begin{verbatim}
if not (set\_exists subcomp\_list set_1 main\_tree =
  set_2 = set\_add subcomp\_list set_1 \in
  sub\_tree_1 = tree\_create subcomp\_list conn\_list subcomp\_list \in
  subcomp\_list_2 = traverse\_connection\_list conn\_list subcomp\_list in
  sub\_tree_2 = traverse\_subcomponent\_list subcomp\_list_2 conn\_list set_2 sub\_tree_1 \in
  tree\_add\_child sub\_tree_2 main\_tree
  else main\_tree
\end{verbatim}

Listing 7 The \textit{traverse\_subcomponent\_list} semantic function.

\begin{verbatim}
traverse\_subcomponent\_list : Integer \times List \times
  List \times Set \times Tree \rightarrow Tree
\end{verbatim}

\begin{verbatim}
traverse\_subcomponent\_list inst\_index subcomp\_list
  conn\_list set tree_1 =
  if inst\_index < (list\_size subcomp\_list) then
    system (state, symbol\_table, inst\_list,
      trans\_list) = list\_get inst\_index subcomp\_list in
    sub\_tree_2 = traverse\_transition\_list trans\_list
      inst\_index subcomp\_list conn\_list set tree_1
      in traverse\_subcomponent\_list (inst\_index + 1)
      subcomp\_list conn\_list set tree_2
  else tree_1
\end{verbatim}

4.2 Tree Evaluation

When the state space tree of Section 4.1 has been generated, it becomes evaluated against the CTL property specification. The \textit{evaluate\_children} (List. 10), and \textit{evaluate\_tree} (List. 11) semantic functions call each other alternately. Initially, \textit{evaluate\_tree} is called with the root node; it calls \textit{evaluate\_children} for its children, which in turn call \textit{evaluate\_tree} for each of the children. These alternated calls continue until the property specification has been satisfied, or a leaf in the tree has been reached.

The \textit{evaluate\_children} traverses the children of the root node of a tree. If there are no children, we have reached a leaf of the tree. Different values are returned, depending on the \textit{depth} operator. In case of the \textit{global} operator, the property has to hold for each node on the path from the root node to the leaf. Therefore, the \textit{and} operator is applied to the node property values, and \textit{true} is returned at the end of the path. In case of the \textit{eventually} operator, it is enough that one property holds for the path from the root node to the leaf node. Therefore, the \textit{or} operator is applied to the node property values, and \textit{false} is returned at the end of the path.
### Listing 8

The `traverse_transition_list` semantic function.

```
traverse_transition_list : List × Integer × List × List × Set × Tree → Tree

traverse_transition_list trans_list inst_index subcomp_list conn_list set tree1 =
  if (list_size trans_list) > 0 then
    let (head, tail) = list_split trans_list in
    let tree2 = execute_transition head inst_index
      subcomp_list conn_list set tree3 in
    traverse_transition_list tail inst_index
      subcomp_list conn_list set tree2
  else tree1
```

If the root node of the tree has one child, we simply evaluate it by calling `evaluate_tree`. However, if it has more than one child, we need to examine the quantifier. In case of the `all` quantifier, the property has to hold for all child nodes, so we apply the `and` operator between the property value of the first child node and the evaluation of the rest of the children. In case of the `exists` quantifier, the property has to hold for only one of the children, meaning that we instead apply the `or` operator.

The `evaluate_tree` semantic rule evaluates the property of the root node of the tree, and compares it with the children. If we assume the `global` operator, the property has to hold for the root node, and all the nodes on the path to the leaf nodes. In case of the `eventually` operator, it is enough if the property holds for one of them.

The `evaluate_node` semantic rule calls `evaluate_node`, which is relatively simple, and therefore has been omitted due to space limitations.

#### Example 1

Let us investigate the AADL model of List. 12 and 13 (originally introduced in [5]). There are two subcomponents: `subsystem1` and `subsystem2`. For each subcomponent, `traverse_transition_list` traverses the transactions and, for each transition that is ready to be taken, it calls `execute_transition`. Finally, `execute_transition` calls `generate_tree` recursively, with the new child node as parameter, in order to attach child nodes recursively. That is, each taken transition represents a new state space that is dealt with by `generate_tree`, as it has been the initial state space. This call chain continues until no more transitions are ready to be taken, or until a previous state space reoccurs (see Fig. 1 for an illustration of the process.

5 Related Work

The approach that we feel is closest to ours is of Őlveczky et al. [10]. The authors have defined a translational semantics from AADL into their object-oriented language Maude, which includes components, port connections, and
Listing 9 The `execute_transition` semantic function.

```maude
execute_transition : Value × Integer × List × List × Set × Tree → Tree
execute_transition trans_value inst_index subcomp_list1
  conn_list set main_tree =
  let transition (source_state, guard_expr,
    target_state, action_list) = trans_value in
  let record1 = table_get inst_index subcomp_list in
  let system (state, symbol_table1, init_list,
    trans_list) = record1 in
  let (boolean is_guard, symbol_table2) =
    evaluate guard_expr symbol_table1 in
  if (state = sourceState) and is_guard then
    let symbol_table3 = traverse_action_list
      init_list symbol_table2 in
    let record2 = system (target_state,
      symbol_table3, init_list, trans_list)
    let subcomp_list2 = list_set inst_index
      record2 subcomp_list1 in
    generate_tree subcomp_list2 conn_list
      set main_tree
  else main_tree
```

(a) Initial State         (b) State 2         (c) State 3         (d) State 4         (e) Final State

Fig. 1: The Main System States

the Behavior Annex. The AADL components and their subcomponent instances are translated into Maude classes and objects. Maude is capable of simulations, and model checking of Linear Temporal Logic (LTL) [11] for embedded system models. However, the authors have chosen an AADL subset that differs from the subset of this paper.

An approach that is also close to ours is the formal semantics defined by Bozzano et al. [12]. It is centered on the concept of components. For each component, its type, interface, and implementation are given. The component interaction is described by a finite state automaton [8]. Their work includes model checking. However, it is centered around detection of errors, as opposed to the semantics of this paper that focuses on system behavior.

Another interesting approach is proposed by Yang et al. [13]. The authors introduce a formal semantics for the AADL Behavior Annex using Timed Abstract State Machine (TASM) [14]. The authors give the semantics of the AADL default execution model, and formally define some aspects of the Behavior Annex. In their translation, each behavior annex is mapped into a TASM main machine.
Listing 10 The \texttt{evaluate\_children} semantic function.

\begin{verbatim}
\textbf{evaluate\_children} : TreeProp \times \text{List} \times \text{WidthOp} \times \text{DepthOp} \rightarrow \text{Boolean}
\textbf{evaluate\_children}\ TP\ child\_list\ quantifier\ operator =
\textbf{case} (\text{list\_size}\ child\_list)\ of
  \textit{0} \Rightarrow \textbf{case}\ operator\ of
                           \textit{global} \Rightarrow \textbf{true}
                          \textit{eventually} \Rightarrow \textbf{false}
  \textit{1} \Rightarrow \textbf{evaluate\_tree}\ TP\ (\text{list\_get}\ 0\ child\_list)
                          \textbf{quantifier}\ operator
  \textit{default} \Rightarrow \textbf{let}\ (head, tail) = \text{list\_split}
                          \text{child\_list}\ in
                          \textbf{case}\ quantifier\ of
                          \textit{all} \Rightarrow (\textbf{evaluate\_tree}\ TP\ head
                          \textbf{quantifier}\ operator)\ \textbf{and}
                          (\textbf{evaluate\_children}\ TP\ tail
                          \textbf{quantifier}\ operator)
                          \textit{exists} \Rightarrow (\textbf{evaluate\_tree}\ TP\ head
                          \textbf{quantifier}\ operator)\ \textbf{or}
                          (\textbf{evaluate\_children}\ TP\ tail
                          \textbf{quantifier}\ operator)
\end{verbatim}

However, even though TASM is a user-friendly and powerful simulation tool, it does not support model checking. Instead, they propose further mapping of the TASM state machine into UPPAAL [15].

We finally mention Abdoul et al. [16], who present an AADL model transformation, and provide a formal model for model checking activities, covering the three main aspects: structure, behavior description, and execution semantics. The authors extend the AADL meta-model in order to improve the system behavior, and they define a translation semantics into the IF language [17], which is a language for simulation of systems and processes. However, the system behavior is not defined in the Behavior Annex, but rather in the IF internal format.

6 Conclusions and Further Work

In this paper, we have presented a formal analysis framework, consisting of a denotational semantics for a subset of AADL and its Behavior Annex, and an implementation of the semantics in Standard ML. The framework is completed by our recently developed graphical Eclipse-based tool, ABV, [5], in which one can model-check AADL descriptions, against CTL properties, in a user-friendly way.

Here, we have given a precise meaning, in denotational style, to a subset of AADL and its Behavior Annex, with a straightforward implementation in Standard ML. This contribution provides an expressive enough formal framework for the formalization of the AADL constructs that we have looked at. An advantage
Listing 11 The evaluate_tree semantic function.

```plaintext
evaluate_tree : TreeProp × Tree × WidthOp × DepthOp → Boolean

evaluate_tree PS tree quantifier operator =
  case operator of
    global ⇒ let (single subProp) = PS in
      (is_true (evaluate_prop_spec subProp tree))
    and (evaluate_children PS
      (tree_get_children tree) quantifier operator)
    | eventually ⇒ let (single subProp) = PS in
      (is_true (evaluate_prop_spec subProp tree))
    or (evaluate_children PS
      (tree_get_children tree) quantifier operator)
```

Listing 12 The Main System.

```plaintext
system implementation MainSystem . impl
subcomponents
  subsystem1 : system Subsystem1;
  subsystem2 : system Subsystem2;
connections
  event port subsystem1 . CriticalLeave -> subsystem2 . CriticalEnter;
  event port subsystem2 . CriticalLeave -> subsystem1 . CriticalEnter;
end MainSystem . impl;
```

of our approach is the fact that the implementation in Standard ML maps the elements of the semantic model straightforwardly.

There are several ways to continue the work of this paper. One obvious approach is to optimize the algorithms behind the semantics when it comes to state space generation and property specification evaluation. It is possible to evaluate the state space "on the fly", that is, the evaluation should take place during the state space generation. Such a technique has already proven efficient in other model-checking tools, including SPIN and UPPAAL.

Another interesting extension of the semantics is adding time annotations to the transitions, in order to perform real-time model checking.

Acknowledgment

This research work was partially supported by Swedish Research Council (VR), and the Swedish Foundation for Strategic Research via the strategic research center PROGRESS.

References

Listing 13 The Subsystem.

system Subsystem1
features
  CriticalEnter: in event port;
  CriticalLeave: out event port;
annex SubsystemAnnex1
{ **
  initializations
    CriticalLeave !;
  states
    Waiting : initial state;
    Critical : state;
  transitions
    Waiting → [CriticalEnter?] → Critical;
    Critical → [true] → Waiting
    { CriticalLeave !; }
  **};
end Subsystem1;