A FORMAL ANALYSIS FRAMEWORK FOR AADL

Stefan Björnander¹, Cristina Seceleanu², Kristina Lundqvist², Paul Pettersson²

¹CrossControl AB Kopparlundsvagen 14, 721 30 Vasteras, Sweden
²School of Innovation, Design, and Engineering Malardalen University, Box 883, 721 23 Vasteras, Sweden

Received September 30, 2011

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 fulfills 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 (SAE¹), 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.

¹SEA 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 semantics 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. See Björnander et al. [5] for a description. 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 that contains the following:

- A denotational semantics defining a subset of AADL and its Behavior Annex that also include model checking of properties defined in CTL. See Björnander et al. [9] for a complete specification of the semantics.
- The semantics implemented in Standard ML, and a parser translating a model defined in AADL and its Behavior Annex as well as a subset of CTL property specification into a format in Standard ML suitable as input.
- The ABV model checker 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 rest of this paper is organized as follows: Section 2 is preliminaries; describing, among other things, the syntax of a subset of AADL and its Behavior Annex. Section 3 defines the information gathering part of the denotational semantics. In Section 4, verification by model checking techniques is described. Finally, Section 5 discusses related work before concluding the paper in Section 6 with conclusions and further work.
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 input and output ports of the subcomponents, not the systems. The syntax of our AADL subset is given in Listing 1.

In order to increase the expressiveness of AADL, it is possible to add annexes. One of them is the Behavior Annex [6, 7] that 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 Listing 2.

CTL is a branching-time temporal logic, that is, it models time 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; among them, 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 holds the values. See Björnander et al. [9] for their complete semantic definitions.

Listing 1. The AADL syntactic rules.
+ one or ore, * zero or more, ? zero or more

```
Model ::= System+ SystemImpl
System ::= system Identifier Features? Annex? end ;
Features ::= features Feature+
Feature ::= Identifier : in event port;
   \ Identifier : out event port;
SystemImpl ::= system implementation Identifier .
   Identifier Subcomponents? Connections
   end ;
Subcomponents ::= subcomponents Subcomponent+
Subcomponent ::= Identifier : system Identifier ;
Connections ::= connections Connection
Connection ::= event port Identifier . Identifier ->
   Identifier . Identifier ;
```

StateVariables ::= state variables StateVariable+
StateVariable ::= Identifier : integer ;
StateVariable ::= states State+
State ::= Identifier : initial state ;
| Identifier : state;
Initializations ::= initializations Action+
Transitions ::= transitions Transition+
Transition ::= Identifier - [ Expression ] -> Identifier ;
| Identifier - [Expression ] -> Identifier 
{ Action+ }
Action ::= Identifier := Expression ;
| Identifier !;
Expression ::= Identifier
| Expression ArithmeticOperation Expression
ArithmeticOperation ::= + | - | * | / |

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 Listing 4 and 5). The denotational semantics consists of the mathematical models of meanings (model [[S SI]] in Listing 4 and system [[S, S]] in Listing 5) and the corresponding semantics functions, respectively (model : Model → Table in Listing 4 and system : System → Table in Listing 5).

In the approach we have chosen, 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 (S, s0, I, Var, P_in, P_out, T) where S is a non-empty finite set of states and s0 ∈ S is a compulsory initial state. Var is a possible empty finite set of state variables. P_in and P_out are the possible empty finite sets of input and output ports, respectively. I ⊆ (Var × Expr) ∪ P_out is a possible empty set of initializations. T ⊆ (S × Expr) ∪ P_out is a possible empty set of transitions, where A ⊆ (Var × Expr) ∪ P_out is a possible empty action set and Expr is made up by 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 Listing 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:

- The system table holds the systems of the model. The information of each system is stored in the tuple \((\text{state, symbol-table, init-list, trans-list})\), where \(\text{state}\) is the current state of the annex (initialized to zero, representing the initial state), \(\text{symbol-table}\) holds the input and output ports of the system as well as states and state variables of the annex, \(\text{init-list}\) holds the list of initializations, and \(\text{trans-list}\) 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 Listing This information is originally stored for each system and copied to the subcomponents instantiating the systems.

For each syntactic rule in 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 limitation of space, we confine our self to showing the \textit{model} (Listing 4) and \textit{system} (Listing 5) rules in this paper. 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, in such that each morphism of the 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 semantic rules were 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. Generation

In this section, we generate the state space tree that is initially made up of one single node holding the initial state space; that is, the subcomponent list in its initial state. Then traverse-subcomponent-list (Listing 7) traverses the subcomponents and for each subcomponent traverse_transition_list (Listing 8) traverses the transitions. For each transition that can be taken, execute_transition (Listing 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 generate^tree (Listing 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 becomes aborted if a previous state space reoccurs.
Listing 6. The \texttt{generate} \_\texttt{tree} semantic function

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

\begin{verbatim}
generate\_tree subcomp\_list conn\_list set, main\_tree =
\textbf{if not} (set.exists subcomp\_list set, \textbf{then})
\textbf{let} set2= set.add subcomp\_list set, \textbf{in}
\textbf{let} sub\_tree1 = tree.create subcomp\_list in
\textbf{let} subcomp\_list2 = traverse\_connection\_list
\textbf{conn\_list subcomp\_list in}
\textbf{let} sub\_tree2 = traverse\_subcomponent\_list
\textbf{subcomp\_list2 conn\_list set, sub\_tree1 in}
\textbf{tree.add\_child sub\_tree2 main\_tree}
\textbf{else} main\_tree
\end{verbatim}

Listing 7. The \texttt{traverse} \_\texttt{subcomponent\_list} semantic function

\begin{verbatim}
traverse\_subcomponent\_list : Integer \times List \times
List \times Set \times Tree \rightarrow Tree
traverse\_subcomponent\_list ints\_index subcomp\_list
\textbf{conn\_list set tree}, =
\textbf{if} inst\_index < (list.size subcomp\_list) \textbf{then}
\textbf{let} system (state, symbol\_table, init\_list,
trans\_list) = list.get inst\_index subcomp\_list in
\textbf{let} tree2 = traverse\_transition\_list trans\_list
\textbf{inst\_index subcomp\_list conn\_list set tree},
in traverse\_subcomponent\_list (inst\_index + 1)
\textbf{subcomp\_list conn\_list set tree2}
\textbf{else} tree1,
\end{verbatim}

4.2 Evaluation

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

The \texttt{evaluate} \_\texttt{tree} 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 of the \texttt{depth} operator.
In case of the global operator, the property has to hold for each node on the path from the root node to the leaf. Therefore, the and operator is applied to the node property values, and true is returned at the end of the path. In case of the eventually operator, it is enough that one property holds for the path from the root node to the leaf node. Therefore, the or operator is applied to the node property values and false is returned at the end of the path.

Listing 8. The `traverse_transition_list` semantic function

```latex
traverse_transition_list : List × Integer × List × List × Set × Tree \rightarrow Tree

traverse_transition_list trans_list ints_index subcomp_list conn_list set tree₁ =

\begin{align*}
\text{if } (\text{list.size } trans\text{.list}) > 0 \text{ then} & \quad \text{let } (\text{head, tail}) = \text{list_split trans\text{.list}} \text{ in} \\
\text{let } tree₂ = \text{execute_transition head inst_index} & \quad \text{subcomp\text{.list} conn\text{.list} set tree₁, in} \\
\text{traverse_transition_list tail inst_index} & \quad \text{subcomp\text{.list} conn\text{.list} set tree₂, in} \\
\text{else } tree₁, \quad \text{end} & \end{align*}
```

If the root node of the tree has one child, we simple 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, why 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, why 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. In case of 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` semantics rule calls `evaluate_node`, which is relative simple and therefore has been omitted due to space limitations.

**Example 1.** Let us investigate the AADL model of Listing 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, 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 transitions represent a new state space that is dealt with by `generate-tree` as it was the initial state space. This call chain continues until no more transitions are ready to be taken or until a previous state space reoccurs (Fig. 1) for an illustration of the process.
5. RELATED WORK

The approach we feel is closest to ours is Olveczky et al. [10]. The authors have defined a translational semantics from AADL into their object-oriented language Maude, which includes components, port connections, and the Behavior Annex.

Listing 9. The execute_transition semantic function

```
execute_transition : Value × Integer × List × List × Set × Tree → Tree
execute_transition trans_value ints_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_list1 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_table,
                           symbol_table3, init_list, trans_list) in
    let subcomp_list2 = list_set inst_index
                           record2 subcomp_list1 in
    generate_tree subcomp_list2 conn_list
        set main_tree
  else main-tree
```
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] on embedded systems. However, they 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 Boz-zano 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 Yang et al. [13]. The authors introduce a formal semantics for the AADL Behavior Annex using Timed Abstract State Machine (TASM) [14]. They 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. 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] that presents an AADL model transformation providing a formal model for model checking activities and covers the three 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.

Listing 10. The evaluate_children semantic function

```plaintext
evaluate_children : TreeProp × List × WidthOp × DepthOp → Boolean

evaluate_children TP child_list quantifier operator =

  case (list_size child_list) of
  0 => case operator of
     global => true
   | eventually => false
  1 => evaluate_tree TP (list.get 0 child.list)
      quantifier operator

  | default => let (head, tail) = list.split
       child.list in

  case quantifier of
   all => (evaluate_tree TP head
        quantifier operator) and
     (evaluate_children TP tail
        quantifier operator)
   | exist => (evaluate_tree TP head
```
quantifier operator) or
(evaluate_children TP tail
quantifier operator)

Listing 11. The evaluate_tree semantic function

\[
\text{evaluate_tree} : \text{TreeProp} \times \text{Tree} \times \text{WidthOp} \times \text{DepthOp} \rightarrow \text{Boolean}
\]

\[
\text{evaluate_tree PS tree quantifier operator} =
\]
\[
\text{case operator of}
\]
\[
global => \text{let (single subProp) = PS in}
\]
\[
(is\_true (evaluate\_prop\_spec subProp tree))
\]
\[
\quad \text{and (evaluate\_children PS}
\]
\[
(tree\_get\_children tree) \text{ quantifier operator})
\]
\[
| \text{eventually} => \text{let (single subProp) = PS in}
\]
\[
(is\_true (evaluate\_prop\_spec subProp tree))
\]
\[
\quad \text{or (evaluate\_children PS}
\]
\[
(tree\_get\_children tree) \text{ quantifier operator})
\]

Listing 12. The main System.

\[
\text{system implementation MainSystem.impl}
\]
\[
\text{subcomponents}
\]
\[
\text{subsystem1: system Subsystem1;}
\]
\[
\text{subsystem2: system Subsystem2;}
\]
\[
\text{connections}
\]
\[
\text{event port subsystem1.CriticalLeave \rightarrow subsystem2.CriticalEnter;}
\]
\[
\text{event port subsystem2.CriticalLeave \rightarrow subsystem1.CriticalEnter;}
\]
\[
\text{end MainSystem.impl;}
\]

Listing 13. The Subsystem.

\[
\text{system Subsystem1}
\]
\[
\text{features}
\]
\[
\text{CriticalEnter: in event port ;}
\]
\[
\text{CriticalLeave: out event port ;}
\]
\[
\text{annex SubsystemAnnex1}
\]
Listing 13. The Subsystem.

```plaintext
system Subsystem1

features

CriticalEnter: in event port ;
CriticalLeave: out event port ;

annex SubsystemAnnex1

{**

initializations
  CriticalLeave ! ;

states
  Waiting : initial state ;
  Critical: state ;

transitions
  Waiting – [CriticalEnter?] \rightarrow Critical;
  Critical – [true] \rightarrow Waiting
    {CriticalLeave !;}

**};
end Subsystem1;
```
6. CONCLUSION AND FURTHER WORK

In this paper, we have presented a formal analysis framework including a de-notational semantics for a subset of AADL and its Behavior Annex [9], and an implementation of the semantics in Standard ML. The framework is completed by our recently developed graphical Eclipse-based tool [5] that performs model checking of CTL properties, in a user-friendly way.

We have given precise meaning in denotational style for 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 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 taking place during the state space generation. A technique that has proven efficient in other model-checking tools, including SPIN and UPPAAL.

Another interesting extension of the semantics is to add time annotation 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


Corresponding author:

Stefan Björnander
CrossControl AB Kopparlundsvagen 14, 721 30 Vasteras, Sweden
Email: stefan.bjornander@crosscontrol.com