ABV – A Verifier for the Architecture Analysis and Design Language (AADL)

Stefan Björnander, Cristina Seceleanu, Kristina Lundqvist, and Paul Pettersson
School of Innovation, Design, and Engineering
Mälardalen University, Box 883, 721 23 Västerås, Sweden
Email: {stefan.bjornander, cristina.seceleanu, kristina.lundqvist, paul.pettersson}@mdh.se

***Abstract*—Designing and developing mission-critical embedded systems is challenging, especially due to additional platform constraints regarding timing and computational resources. The development process of embedded systems should include verification techniques already at the architecture design phase, to provide evidence that a system's architecture fulfills its requirements. The Architecture Analysis and Design Language (AADL) is used to model the system's architecture. Among others, the language contains a Behavior Annex, for describing the behavior of an AADL model, at an abstract level.**

In this paper, we present a verification tool, called ABV, tailored for AADL models with a behavioral annex. Given an architecture defined in AADL and its behavior specified in the associated language, our tool model-checks the latter against the requirements specified in Computation Tree Logic (CTL). ABV is based on AADL's formal denotational semantics implemented in Standard ML, and is encapsulated into an Eclipse plug-in based on the OSATE platform. The tool has been applied on the Production Cell case study, which is briefly described in the paper.

***Index Terms*—Model Checking, Verification, AADL, Behavior Annex, CTL, OSATE, Denotational Semantics.**

I. Introduction

Mission-critical embedded systems play a vital role in aerospace applications, air traffic control, and railway signaling, to mention a few. Design and development of such systems is challenging because, e.g., the fulfillment of real time requirements and resource constraints need to be proven in the development process. Of high practical interest is the architecture design phase, since the timing behavior and resource consumption of systems depend heavily on the chosen system architecture. Consequently, the development process for embedded systems should include verification techniques in the architecture design phase, to provide evidence that a system architecture fulfills its requirements.

We have chosen to study the Architecture Analysis and Design Language (AADL) [1] and its Behavior Annex [2], [3], [4], due to AADL's rich specification language and its industrial use for the development of embedded systems in, e.g., the automotive and avionics area. AADL is a large language intended for designing both system software and hardware. The architectural language is a standard of the Society of Automotive Engineers (SAE[1]), and it is based on MetaH [5] and the Unified Modeling Language (UML) [6].

[1]The Society of Automotive Engineers is presented at http://www.sae.org.

In this paper, we introduce and describe a novel system architecture verifier, called ABV[2] (**A**ADL and the **B**ehavior Annex **V**erifier), which performs model checking of embedded system models defined in AADL and its Behavior Annex. The model checking is carried out against requirements formulated in Computation Tree Logic (CTL) [7]. ABV is an Eclipse plug-in based on the Open Source AADL Tool Environment (OSATE[3]) platform, which guarantees seamless interaction with the Eclipse environment. Moreover, ABV is based on an ML implementation of the formal denotational semantics that unambiguously defines a subset of AADL and its Behavior Annex [8], [9].

The intended users of ABV are engineers working with system design; more specifically, engineers modeling AADL systems in the Eclipse environment, who want to check different system properties about the system. By employing our tool, it should be possible that engineers find flaws already in the early design phase, and by that avoid not finding flaws until later development phases.

ABV accepts a subset of AADL including *system*, *system implementation*, *features*, *subcomponents*, and *connections* as well as the Behavior Annex. The chosen subset contains all necessary constructs needed for generic modeling.

Although several tools for analyzing AADL have been developed (see Section VI), they do not integrate a model checker for AADL and its Behavior Annex based on a sound underlying semantics within the OSATE platform.

In our approach, we combine powerful model-checking techniques with an underlying formal denotational semantics and a user-friendly graphical interface. We also describe how ABV has been applied on the Production Cell case study.

The salient features of ABV are:

- An underlying formal denotational semantics for AADL and its Behavior Annex [8], [9].
- An implementation of the denotational semantics in Standard ML.
- An Eclipse plug-in based on the OSATE platform, offering an intuitive and user-friendly graphical interface, well integrated with other OSATE-based products.
- A trace file generator, listing a shortest path from the

[2]ABV is available at http://www.idt.mdh.se/~sbr02/ABV.
[3]The OSATE platform is described and available for downloading at http://-www.aadl.info/aadl/currentsite/tool/osate.html.

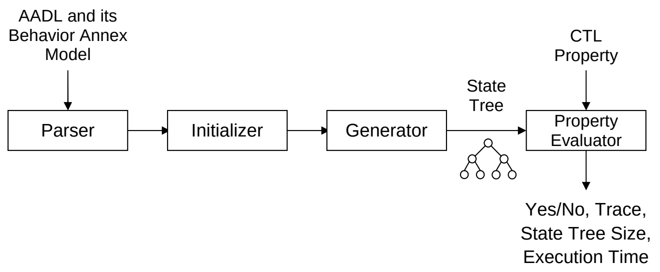

AADL and its Behavior Annex Model
Parser
Initializer
Generator
State Tree
CTL Property
Property Evaluator
Yes/No, Trace, State Tree Size, Execution Time

Figure 1: The architecture of the ABV Kernel.

initial state of the model to a state satisfying the condition of the CTL requirement.

The rest of this paper is organized as follows. Section II gives an overview of ABV. Section III describes ABV's input, whereas Section IV describes the graphical interface. We apply ABV on the Production Cell case study in Section V. In Section VI we discuss related work, before concluding the paper in Section VII.

II. ABV Tool Overview

The main purpose of ABV is to perform model checking by verifying CTL properties of system models defined in AADL and its Behavior Annex. ABV provides an intuitive and user-friendly graphical environment. The user inputs the source code of the model, as well as the CTL property specification, and the verifier displays the yes/no analysis result.

Denotational semantics is an approach to formalize the meaning of programming languages by constructing mathematical objects (called denotations), which describe the meaning of expressions from the languages. The main benefit with denotational semantics is that it is based on a rigorous mathematical foundation and is built on the same principles as functional programming languages, such as Standard ML.

Such benefits determined us to define an underlying denotational semantics for AADL and its Behavior Annex [2], [3], [4], and consequently implement the verifier in Standard ML.

The ABV tool consists of the following modules (see Figure 1): the *Parser*, *Initializer*, *Generator* and *PropertyEvaluator*. The *Parser* module parses, type checks and stores the model. The *Initializer* module initializes the model; that is, the initializations of state variables and output signals are executed. The *Generator* module generates the state space that is later evaluated against the CTL property specification. Finally, the *PropertyEvaluator* module traverses the state space and eventually decides on the truth value of the property.

The root node of the generated state space represents the initial state of the system (see Listing 1). For each possible transition, a child node representing the new system state (after the transition has been taken) is added to the node. If several transitions are possible, several child nodes are added. Each child node represents the system state after some transition has been taken, respectively. ABV keeps track of all states, such that, if one state is repeated, the state space generation along that path is aborted. After the state space has been generated, it is next evaluated against the CTL property specification.

Listing 1 The State Space Generation Function (pseudo code).

```
function generate_space subcomp_list conn_list
                    logg_set main_space
  for each subcomp in subcomp_list
    for each trans in subcomp.trans_list
      (boolean guard_value, symbol_table$_2$) ←
        evaluate_expression trans.quard
                    subcomp.symbol_table
      if (subcomp.state = trans.source_state) and
          guard_value then
        subcomp.state ← subcomp.target_state
        subcomp.symbol_table ← execute_actions
              trans.action_list subcomp.symbol_table$_2$
        if not (logg_set.exists subcomp_list) then
          logg_set.add subcomp_list
          subcomp_list ← execute_connections conn_list
                    subcomp_list
          sub_space ← space_create subcomp_list
          generate_space subcomp_list conn_list
                    logg_set sub_space
          main_space.add_child sub_space
  return main_space
```

Listing 2 The ABV Parser Output.

```
val PropSpec = (tree (all, global, (single
(node (not (node (and ((node (ident
("subSystem1", "Critical"))), (node (ident
("subSystem2", "Critical"))))))))));
```

Since the kernel of ABV is written in ML, its input needs to be translated from AADL and its Behavior Annex, as well as CTL, into Standard ML format. The ABV Parser performs the transformation task. It is written in Standard Java, the CUP Parser Generator for Java, and the JLex Lexical Analyzer Generator for Java. For instance, the following safety property specification is translated into the Standard ML code of Listing 2:

```
all global not (subsystem1.Critical and
                subsystem2.Critical)
```

III. Tool Input

ABV takes two inputs: the AADL and its Behavior Annex model, as well as the CTL property specification.

A. The AADL and its Behavior Annex Model

In AADL, there are two types of systems: the *system* that defines the port interface and an optional behavioral annex, and the *system implementation* that defines the subcomponents and the port connections between them. The chosen AADL subset allows at least one system and exactly one system implementation. 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

Listing 3 Subsystem1 from Figure 2.

```
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;
```

made between input and output ports of the subcomponents, rather than the systems.

The Behavior Annex is based on the Abstract State Machine [10] language. It models the following: (i) states, among which one is the initial state, and state variables, which can be initialized; (ii) input and output signals that are connected to the ports of the surrounding system, together with initializations of the output signals, and (iii) the set of state transitions [3], where each transition is equipped with a guard, that is, a boolean expression that has to evaluate to true for the transition to be fired.

Example 1: Figure 2 illustrates a main system holding two subsystems with one critical section each, modeled in AADL and its Behavior Annex. The subsystems communicate through port signals *CriticalLeave* and *CriticalEnter* in order to ensure that they cannot access the critical sections simultaneously; the code is given in Listings 3 and 4. The first subsystem is initialized to trigger the *CriticalLeave* signal, which means that the second subsystem is free to enter its critical section. The second subsystem is similar to the first one, except that the *initializations* part is omitted. Each subsystem starts in the initial state *Waiting*. As soon as it receives the *CriticalEnter* signal from the other subsystem, it enters its critical section. When it leaves the latter, it sends the *CriticalLeave* signal to the other subsystem, in order to allow it to enter its critical section in turn. The main system instantiates the two subcomponents, *subsystem1* and *subsystem2*, and connects them to each other by using the *CriticalLeave* and *CriticalEnter* ports. In Section III-B, we define a CTL property specification for such system.

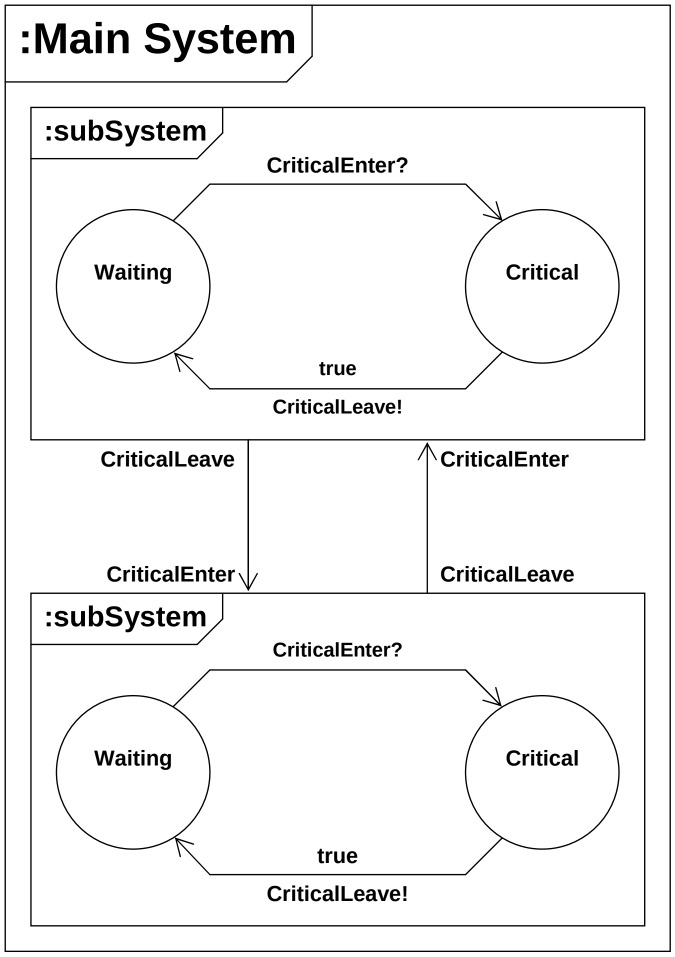


Figure 2: Main System.

Listing 4 The Main System.

```
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;
```

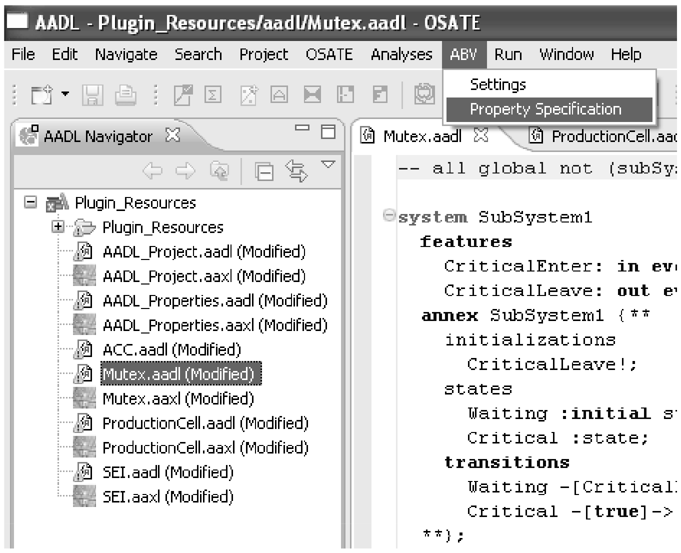
B. The CTL Property Specification

CTL [7] is a branching-time temporal logic, that is, it models time as a tree structure with a non-determined future. There are different paths into the future, and any one of them may be the realized one. There are several quantifiers available in CTL, among them the universal *all* and existential *exists* quantifiers over paths. There are also the path-specific operators *global* and *eventually*. See Table I for their informal definitions.

Syntactically, a property specification includes a path quantification (*all* or *exists*), followed by path-specific operators (*global* or *eventually*), or a regular expression. However, these kinds of expressions can also hold a state or a state variable.

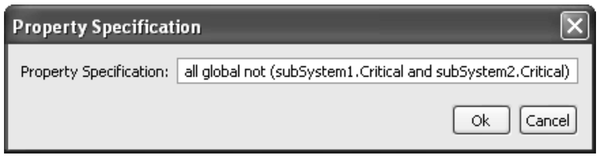
For example 1 described in Section III-A, we formulate below the CTL property that, provided that it holds for the model, guarantees that the two subsystems never reach their critical section simultaneously:

```
all global not (subsystem1.Critical and
                subsystem2.Critical)
```


AADL - Plugin_Resources/aadl/Mutex.aadl - OSATE
File Edit Navigate Search Project OSATE Analyses ABV Run Window Help
Settings
Property Specification
AADL Navigator
Mutex.aadl
ProductionCell.aa
Plugin_Resources
Plugin_Resources
AADL_Project.aadl (Modified)
AADL_Project.aaxl (Modified)
AADL_Properties.aadl (Modified)
AADL_Properties.aaxl (Modified)
ACC.aadl (Modified)
Mutex.aadl (Modified)
Mutex.aaxl (Modified)
ProductionCell.aadl (Modified)
ProductionCell.aaxl (Modified)
SEI.aadl (Modified)
SEI.aaxl (Modified)

```
-- all global not (subSy
system SubSystem1
  features
    CriticalEnter: in ev
    CriticalLeave: out e
  annex SubSystem1 {**
    initializations
      CriticalLeave!;
    states
      Waiting :initial s
      Critical :state;
    transitions
      Waiting -[Critical
      Critical -[true]->
  **};
```

Figure 3: The Graphical Interface.


Property Specification

Property Specification: all global not (subSystem1.Critical and subSystem2.Critical)

Ok Cancel

Figure 4: The CTL Property Specification.

IV. The Graphical Interface

ABV is encapsulated in an Eclipse plug-in on top of the OSATE platform. The plug-in adds the ABV menu with the *Settings* and *Property Specification* items (see Figure 3). The *Settings* item displays an input dialog divided into two parts, where the first part specifies a set of paths needed for ABV to work properly while the second part specifies optional output to be displayed.

The result of model checking a property specification against a model of AADL and its Behavior Annex is a boolean value. ABV also reports the size of the generated state space and the execution time, depending on the user settings in the second part of the dialog. ABV can also be instructed to generate a log file, listing a sequence of states from the initial state to a state satisfying the condition of the property specification (if no state is satisfying the condition, an empty file is generated). See Appendix B for an excerpt.

The *Property Specification* item displays an input dialog in which the user inputs the CTL property specification (see Figure 4).

Table I: CTL.

all ϕ	ϕ must be satisfied for every path.
exists ϕ	ϕ must be satisfied for at least one path.
global ϕ	ϕ must be satisfied for each node on the path from root to leaf.
eventually ϕ	ϕ must be satisfied for at least one node on the path from root to leaf.

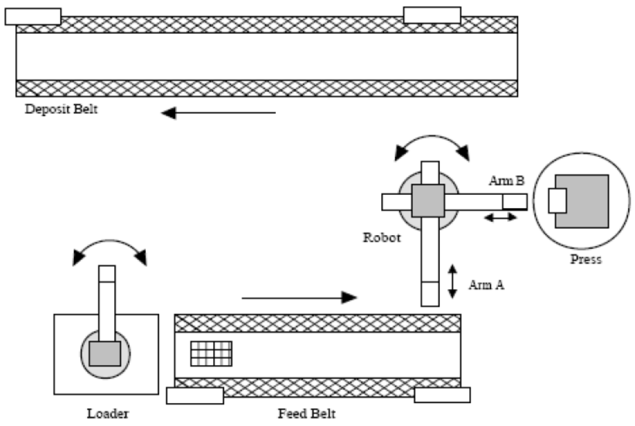


Figure 5: The Production Cell System as presented by Ouimet [12].

V. Case Study

In order to validate ABV, we have performed a case study in which a Production Cell system is modeled in AADL and its Behavior Annex. For the complete source code of the system, we refer the reader to Björnander et al. [8], [9]. Also, see Appendix A for an excerpt. The case study is based on an automated manufacturing system that is modeled on an industrial plant in Karlsruhe, Germany. It was first described by Lewerentz [11]. Ouimet [12] presented it as depicted in Figure 5.

The system is not controlled by a central unit. Instead, the components communicate with each other through port connections. The components work concurrently; when a component is ready to accept a new block, it notifies the preceding component, which in turn acknowledges that is has loaded the block. It also sends a signal acknowledging that the loading location of the component is free.

The Production Cell System is composed of the robot arms *Loader*, *ArmA*, and *ArmB*, the conveyer belts *FeedBelt* and *DepositBelt*, as well as the *Press*. The purpose of the system is to transport blocks through the production cell. They arrive in a crate and the same blocks with bolts attached to them finally become delivered by the deposit belt for further processing. Once a block has been loaded, it is moved through the system. See Figure 6 for a schematic illustration of the system.

In the *DepositBelt* subsystem, we have added the state variable *StoredBlocks* that counts the number of blocks that become transported through the final deposit belt. In the main system of the Production Cell, the *DepositBelt* system has been instantiated into the *depositBelt* subcomponent. In order to make sure that a loaded block always becomes transported through the system, we require that the state variable *storedBlocks* in the *depositBelt* subcomponent is always eventually assigned the value one. This means that a block has gone through the whole system. The requirement can be formally stated in CTL as follows:

```
all eventually depositBelt.StoredBlocks = 1
```

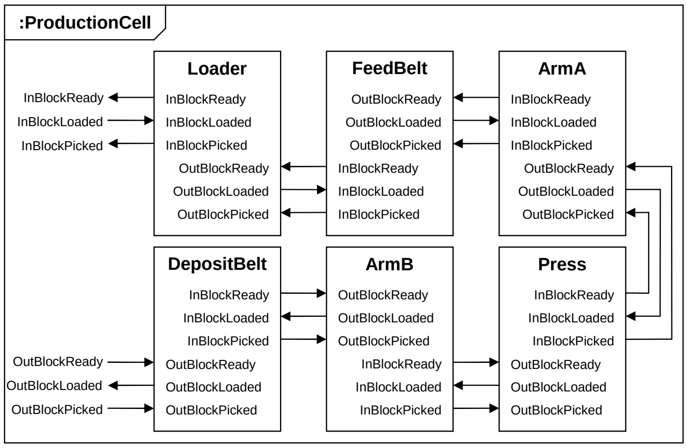
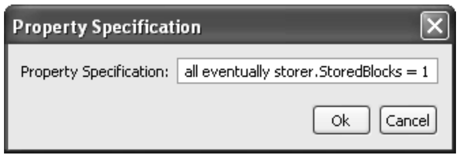


Figure 6: Schematic Description of the Production Cell System.



Property Specification: all eventually storer.StoredBlocks = 1

Figure 7: Property Specification.

As evident from Figure 8, all paths finally lead to a state where *storedBlocks* is equals to one, meaning that the block is always moved through the Production Cell system.

As ABV and also the underlying denotational semantics also support verification of architectural properties, the following CTL specification is an example of such property, which ensures that the *InBlockLoaded* input signal in the *depositBelt* subcomponent is never written twice without being read between the writings:

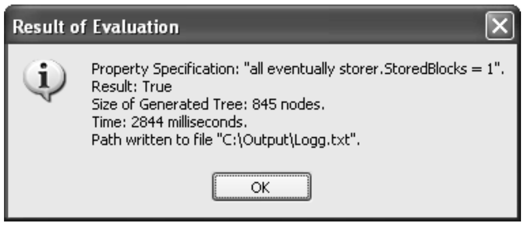
```
all global depositBelt.InBlockLoaded_count <= 1
```

For each input port, a state variable with the same name and the suffix "_count" is stored. It is an integer that becomes incremented each time a signal is sent to the port, and decremented each time it is read. If the variable never exceeds one, we can be sure that once a signal is sent to the port, it is always read before the next signal is sent.

VI. Related Work

Several tools have been developed for simulation and analysis of AADL and its annexes. ADeS[4] (Architecture Description Simula) is a software tool intended for simulation of the behavior of system architectures described with AADL. It is implemented as an Eclipse plug-in based on the OSATE platform, with seamless interaction with other OSATE based tools. It can perform simulation and scheduling of models defined in AADL. However, unlike ABV, it does not perform model checking, and it also lacks a formal underlying analysis.

[4]The AdeS tool is available at http://www.axlog.fr/aadl/ades_en.html.



```
Property Specification: "all eventually storer.StoredBlocks = 1".
Result: True
Size of Generated Tree: 845 nodes.
Time: 2844 milliseconds.
Path written to file "C:\Output\Logg.txt".
```

Figure 8: Result of Evaluation.

BIP (Behavior Interaction Priority) [13] is a language for description and composition of components, as well as for analysis and code generation of models. Chkouri et al. [14] describe a general methodology and an associated tool for translating AADL and its Behavior Annex specifications into BIP. This allows simulation of systems specified in AADL and application of formal verification techniques developed for BIP. The input language of the tool includes the Behavior Annex. Even though BIP supports model checking of component interaction, it does not support model checking at the system level, as ABV does.

The COMPASS[5] project (Correctness, Modeling, and Performance of Aerospace Systems) is an ambitious attempt to fully capture the system under development by applying model-checking techniques. The COMPASS toolset has been developed by Bozzano et al. [15], [16] and is based on a formal semantics. It is a tool that models both the nominal and faulty behavior of AADL, and it has been validated in several case studies. However, in contrast to ABV, it focuses on the error behavior. Moreover, the tool is equipped with a graphical interface that, unlike ABV, is not OSATE-based and hence not integrated with tools based on Eclipse or OSATE.

VII. Conclusion and Future Work

In this paper, we have presented ABV, which is a novel tool designed to perform model checking on systems modeled in AADL and its Behavior Annex, with properties specified in CTL. Our tool comes with a graphical interface as an Eclipse plug-in on top of the OSATE platform, and it has been successfully applied on the Production Cell case study.

The next step on the work of this paper would be to optimize the algorithms of ABV, e.g., the state space generation and the property specification evaluation. It should be possible to evaluate the state space "on-the-fly"; that is, to perform evaluation during the tree state space generation. In that way, the state space generation can be terminated as soon as the value of the property specification has been determined.

Another possible future work is to extend the semantics with timing annotations, to enable modeling of timing and verification of the form, e.g., minimum and maximum time for a property specification to be satisfied.

[5]The COMPASS project is presented at http://compass.informatik.rwth-aachen.de.

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

[1] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The Architecture Analysis and Design Language (AADL): An Introduction," Society of Automotive Engineers, Tech. Rep. CMU/SEI-2006-TN-011, 2006.

[2] The SAE Technical Standards Board, "The annex behavior specification," SAE International, Tech. Rep. AS5506, 2007.

[3] P. Feiler and B. Lewis, "SAE Architecture Analysis and Design Language (AADL) Annex Volume 1," Society of Automobile Engineers, Tech. Rep. AS5506/1, 2006.

[4] R. B. França, J.-P. Bodeveix, M. Filali, J.-F. Rolland, D. Chemouil, and D. Thomas, "The AADL behaviour annex - experiments and roadmap," in *ICECCS*. IEEE Computer Society, 2007, pp. 377–382.

[5] S. Vestal and J. Krueger, "Technical and Historical Overview of MetaH," Honeywell Technology Center, Tech. Rep. MN 55418-1006, 2000.

[6] D. Pilone and N. Pitman, *UML 2.0 in a Nutshell*, 2nd ed. O'Reilly Media, 2005.

[7] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*, 2nd ed. Cambridge University Press, 2004.

[8] S. Björnander, C. Seceleanu, K. Lundqvist, and P. Pettersson, "The architecture analysis and design language and the behavior annex: A denotational semantics," Mälardalen University, Technical Report ISSN 1404-3041 ISRN MDH-MRTC-251/2011-1-SE, January 2011.

[9] S. Björnander, C. Seceleanu, K. Lundqvist, and P. Pettersson, "A Denotational Semantics for the Architecture Analysis and Design Language (AADL)," unpublished.

[10] E. Borger and R. Stark, *Abstract State Machines - A Method for High-level System Design and Analysis*. Springer-Verlag Berlin And Heidelberg Gmbh and Co. Kg, 2003.

[11] C. Lewerentz and T. Lindner, "Production Cell: A Comparative Study in Formal Specification and Verification," *Lecture Notes in Computer Science*, vol. 1009, pp. 388–398, 1995.

[12] M. Ouimet and K. Lundqvist, "Modeling the Production Cell System in the TASM Language," Embedded Systems Laboratory, Massachusetts Institute of Technology, Cambridge, MA, 02139, USA, Tech. Rep. ESL-TIK-00209, 2007.

[13] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in BIP," 2006.

[14] M. Y. Chkouri, A. Robert, M. Bozga, and J. Sifakis, "Translating AADL into BIP - application to the verification of real-time systems," in *MoDELS Workshops*, ser. Lecture Notes in Computer Science, M. R. V. Chaudron, Ed., vol. 5421. Springer, 2008, pp. 5–19. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-01648-6

[15] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, M. Roveri, and R. Wimmer, "A model checker for AADL," in *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. Jackson, Eds., vol. 6174. Springer, 2010, pp. 562–565. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14295-6

[16] M. Bozzano, R. Cavada, A. Cimatti, J.-P. Katoen, and V. Nguyen, "Formal Verification and Validation of AADL Models," in *ERTS*, 2010.

Appendix

A. The Case Study Source Code (excerpt)

```
system Loader
  features
    InFeedBeltReady: in event port;
    OutBlockReady: out event port;
  annex Loader {**
   state variables
     LoadedBlocks :integer;
   initializations
     LoadedBlocks := 0;
   states
     Waiting :initial state;
     Loading :state;
   transitions
      Waiting -[(LoadedBlocks < 1) and
                (InFeedBeltReady?)]-> Loading;
      Loading -[true]-> Waiting {OutBlockReady!;
               LoadedBlocks := LoadedBlocks + 1;}
  **};
end Loader;

...

system implementation ProductionCell.impl
  subcomponents
    loader: system Loader;
    feedBelt: system FeedBelt;
    beltToPress: system BeltToPress;
    press: system Press;
    pressToBelt: system PressToBelt;
    depositBelt: system DepositBelt;
    storer: system Storer;
  connections
    event port feedBelt.InFeedBeltReady ->
               loader.InFeedBeltReady;
    event port loader.OutBlockReady ->
               feedBelt.InBlockReady;
    event port beltToPress.InArmReady ->
               feedBelt.InArmReady;
    event port feedBelt.OutBlockReady ->
               beltToPress.InBlockReady;
    event port press.PressReady ->
               beltToPress.PressReady;
    event port beltToPress.OutBlockReady ->
               press.InBlockReady;
    event port pressToBelt.OutArmReady ->
               press.OutArmReady;
    event port press.OutBlockReady ->
               pressToBelt.InBlockReady;
    event port depositBelt.OutFeedBeltReady ->
               pressToBelt.OutFeedBeltReady;
    event port pressToBelt.OutBlockReady ->
               depositBelt.InBlockReady;
    event port storer.StorerReady ->
               depositBelt.StorerReady;
    event port depositBelt.OutBlockReady ->
               storer.InStorerBlockReady;
end ProductionCell.impl;
```

B. The Generated Log File (excerpt)

```
0:
Transition: loader.Waiting -> loader.Loading
State: loader = Loading, LoadedBlocks = 0
       feedBelt = NoBlock_MotorOff
       beltToPress = MagnetOff_AtBelt_Retracted
       press = Waiting
       pressToBelt = MagnetOff_AtPress_Retracted
       depositBelt = NoBlock_MotorOff
       storer = Waiting, StoredBlocks = 0

...

24:
Transition: storer.Storing -> storer.Waiting
State: loader = Waiting, LoadedBlocks = 1
       feedBelt = BlockAtEnd_MotorOff
       beltToPress = MagnetOff_AtBelt_Retracted
       press = Waiting
       pressToBelt = MagnetOff_AtPress_Retracted
       depositBelt = BlockAtEnd_MotorOff
       storer = Waiting, StoredBlocks = 1
```