Timed Simulation of Extended AADL-Based Architecture Specifications with Timed Abstract State Machines

Stefan Björnander[1], Lars Grunske[2], and Kristina Lundqvist[3]

[1]School of IDE, Mälardalen University, Box 883, 72123 Västerås, Sweden, Phone: +46 21 101 689, stefan.bjornander@mdh.se
[2]Faculty of ICT, Swinburne University of Technology Hawthorn, VIC 3122, Australia, Phone: +61 3 9214 5397, lgrunske@swin.edu.au
[3]School of IDE, Mälardalen University, Box 883, 72123 Västerås, Sweden, Phone: +46 21 101 428, kristina.lundqvist@mdh.se

Abstract. The Architecture Analysis and Design Language (AADL) is a popular language for architectural modeling and analysis of software intensive systems in application domains such as automotive, avionics, railway and medical systems. These systems often have stringent real-time requirements. This paper presents an extension to AADL's behavior model using time annotations in order to improve the evaluation of timing properties in AADL. The translational semantics of this extension is based on mappings to the Timed Abstract State Machines (TASM) language. As a result, timing analysis with timed simulation or timed model checking is possible. The translation is supported by an Eclipse-based plug-in and the approach is validated with a case study of an industrial production cell system.

Keywords: AADL, Behavior Annex, TASM, Translation.

1 Introduction

Time-critical embedded systems play a vital role in, e.g. aerospace, automotive, air traffic control, railway, and medical applications. Designing such systems is challenging, because the fulfillment of real-time requirements and resource constraints has to be proven in the development process. The architecture design phase is of specific practical interest, as the timing behavior and resource consumption of systems depend heavily on the architecture chosen for these systems. Furthermore, architectural mistakes that cause a system not to fulfill certain real-time requirements are hard to correct in later development phases. As a result, a development process for time-critical embedded systems should include verification techniques in the architecture design phase to provide evidence that a system architecture has the potential to fulfill its real-time requirements [1–3].

In this paper, timed simulations are used for evaluating real-time requirements. The Architecture Analysis and Design Language (AADL) [4] has been chosen, due to the sound specification language and its industrial use for the development of embedded systems in the automotive and avionic area. To allow for a specification of timing behavior in AADL, we have extended AADL's behavior annex [5] with time annotations that provide a minimum and maximum time for each behavioral transition in the model. To provide a formal semantics of these extensions, Timed Abstract State Machines (TASMs) [6] are used as the formal foundation. In detail, this paper describes a translational semantic that maps the extended behavior specifications of AADL into a network of timed abstract state machines. As a result, existing evaluation and verification techniques, such as timed simulations [7] and timed model checking [3] defined for the TASM specification formalism could also be applied to extended AADL specifications. The translation of time-extended AADL specifications into TASM is supported by a tool called AADLtoTASM and the overall approach is validated with a case study of an industrial production cell.

The rest of this paper is organized as follows: Section 2 introduces the running example of a production cell system and gives an overview of the used specification formalisms: AADL with its behavior annex as well as TASM. Section 3 presents the time extension of AADL's behavior annex. Furthermore, the translational semantics and tool support are also presented in this section. The results of timed simulations of the case study are presented in Section 4. Finally, in Section 5 the approach is compared with related work and Section 6 concludes the paper and gives an outlook to future work.

2 Background

2.1 The Production Cell - a Running Example

In order to explain and validate the approach of this paper, a running example of a production cell system is used. This case study is based on an automated manufacturing system which models an industrial plant in Karlsruhe (Germany). The industrial production cell system was first described by Lewerentz and Lindner in [8]. Ouimet et al. defined it in TASM in [9] as depicted in Figure 1.

The overall purpose of the system is to attach two bolts to a metal block. The system is not controlled by a central unit. Instead, the production cell components communicate with each other through ports and bus 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. There is also a signal acknowledging that the loading location of the production cell component is free.

The system is composed of the robot arms *Loader*, *BeltToPress* (Arm A), and *PressToBelt* (Arm B), the conveyer belts *FeedBelt* and *DepositBelt* as well as the *Press*. The system input are sets of blocks arriving in crates and the output are the same blocks with bolts attached to them. Once a block has been loaded, it is "dragged" through the system. See Figure 2 for a schematic description.

To describe the behavior of the production cell components let us look at the loader. While waiting, it is parked at the crate with the magnet turned off.

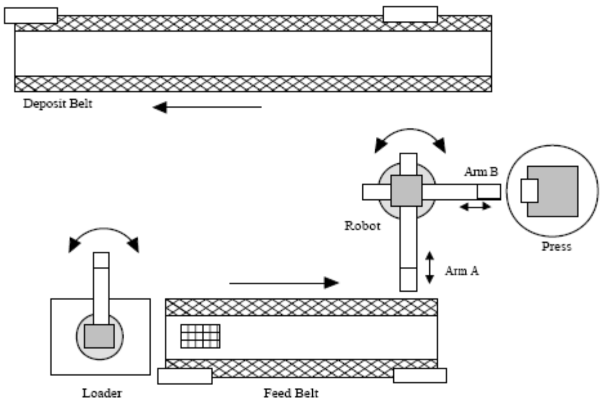

Fig. 1. The Production Cell System as presented in [9].

When it receives a signal from the feed belt that it is ready to receive a new block, the loader turns the magnet on, moves the arm onto the beginning of the feed belt, turns the magnet off, and signals the feed belt that it has loaded a block. The rest of the production cell components work in similar fashions.

2.2 The Architecture Analysis and Design Language

AADL (aadl.info) is a language intended for the design of system hardware and software. It is a Society of Automotive Engineers (www.sae.org) standard based on MetaH [10] and UML 2.0 [11, 12].

The AADL standard [4] includes runtime semantics for mechanisms of exchange and control of data, including message passing, event passing, synchronized access to shared data, thread scheduling protocols, and timing requirements.

AADL can be used to model and analyze systems already in use as well as to design new systems. AADL can also be used in the analysis of partially defined architectural patterns. Moreover, AADL supports the early prediction and analysis of critical system qualities, such as performance, schedulability, and reliability.

In AADL, a model consists of syntactical *elements*. There are three categories of elements. The first category is the application software:

- **Thread**. A thread can execute concurrently and be organized into thread groups.
- **Thread Group**. A thread group is a component abstraction for logically organizing threads or thread groups within a process.
- **Process**. A process is a protected address space whose boundaries are enforced at runtime.
- **Data**. A data component models types as well as static data.

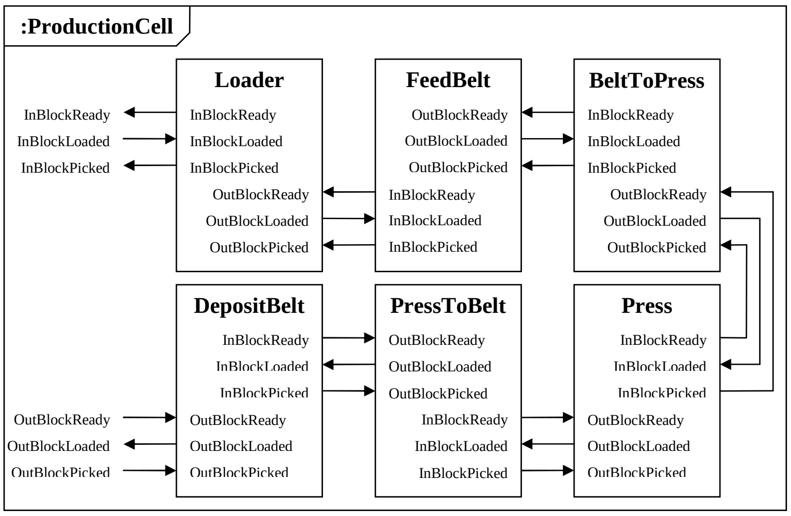

Fig. 2. Schematic Description of the Production Cell System.

– **Subprogram**. A subprogram models a callable block of source code.

The second category is the execution platform (the hardware):

– **Processor**. A processor schedules and executes threads.
– **Memory**. A memory component is used to store code and data.
– **Device**. A device represents sensors and actuators that interface with the external environment.
– **Bus**. A bus interconnects processors, memory, and devices.

The third category contains only one element: the system components. System components can consist of software and hardware components as well as other systems.

Component definitions are divided into *types* holding the public (visible to other components) features, and *implementations* that define the non-public parts of the component.

The components interact through defined interfaces, which consist of directional flows through event and data ports. It is possible to define physical port-to-port connections as well as logical flows through chains of ports.

Listing 1 shows the AADL ProductionCell system of section 2.1 (in AADL, two hyphens introduce a comment).

2.3 The Timed Abstract State Machines

TASM is specification language for reactive real-time systems. It has been developed at the Embedded Systems Laboratory (esl.mit.edu) at the Massachusetts Institute of Technology as a part of the Hi-Five project [13].

Listing 1 The ProductionCell System.

```
system ProductionCell
end ProductionCell;

system implementation ProductionCell.impl
  subcomponents
    loader :system Loader;
    feedBelt :system FeedBelt;
    robot :system Robot;
    press :system Press;
    depositBelt :system DepositBelt;
  connections
    event port feedBelt.FeedBeltReady -> loader.FeedBeltReady;
    event port loader.BlockDropped -> feedBelt.BlockDropped;
    event port feedBelt.BlockPicked -> loader.BlockPicked;
    -- ...
end ProductionCell.impl;
```

Formally, a TASM specification is a pair $< E, ASM >$ [14] where E is the *environment*, which is a pair $E =< EV, TU >$ where EV denotes the *environment variables* (a set of typed variables) and TU is the *type universe*, a set of types that includes real numbers, integer, boolean constants, and user-defined types.

ASM is the machine, which is a triple $< MV, CV, R >$ where MV is the set of read-only *monitored variables*, CV is the set of *controlled variables* (both MC and CV are subsets of EV) and R is the set of *rules* (n, r) where n is a name and r is a rule of the form "**if** C **then** A" where C is an expression that evaluates to an element in BVU and A is an action. It is also possible to attach an else rule on the form "**else** A".

Technically, a TASM specification is made up of *machines*. A specification must hold at least one *main machine* with its set of rules. Beside main machines, it is also possible to define sub machines and function machines. A sub machine accesses[1] the variables of the model and work as a procedure in a programming language, with the difference that it does not accept parameters. A function machine is equivalent to a function in a programming language; it accepts parameters and returns a value. However, it cannot access the global variables of the model.

A transition between two states in a TASM machine can be annotated by a time interval defining the minimal and maximal time to perform the transition.

The TASM Toolset [14] is an integrated development environment, composed of a project manager, an editor, an interpreter, and a simulator. The specification machines can be simulated in the Toolset.

[1] The term *access* is a comprehensive term for *inspect* and *modifiy*.

3 Time Extension of the AADL Behavior Annex

3.1 The AADL Behavior Annex Extension

In order to increase the expressiveness of AADL, it is possible to add *annexes*. One of them is the Behavior Annex [15] that models an abstract state machine [16].

Each component of the model describes its logic by defining a behavior model, which consists of three parts [5]:

- **States**. The states of the machine, one of them is the initial state.
- **Transitions**. The condition for a transition from one state to another (or the same state) is determined by a guard: an expression that evaluates to `true` or `false`. It is also possible to attach a set of actions to be executed when the transition is performed.
- **State Variables with Initializations**. The variables are similar to variables in programming languages. They can be initialized, inspected, and assigned.

In this paper, we add the concept of *time* to the transitions. In addition to the guard, each transition also has a time interval defining the minimal and maximal time for the transition performance. The grammar of the extended behavior model is given in Table 3, with the extensions underlined. The time annotations can be an interval, a single integer value representing both the lower and upper limit of an interval, or the word **null**, representing the absence of a time annotation.

Listing 2 defines the behavior model of the ProductionCell system that has been described earlier in Listing 1. The model is extended in relation to the standard in two ways:

- **Channel Initialization**. When the model starts to execute, a trigger message is sent to the OutBlockReady channel.
- **Time**. When the transitions occur, a time period is recorded. It is one time unit in all four transitions of Listing 2.

3.2 Translation Semantics

The translation to TASM has two phases: analysis and generation. The analysis phase follows the syntax of Table 3 to achieve a full coverage of the language. The aim of this phase is to translate an extended AADL specification into a tuple consisting of a state variable map, a state variable set, a set of state variables to become initialized, the state set, the initial state (it can only be one), and the transaction set. The generation phase takes the tuple defined in the analysis phase as an input and returns for each transition (if present) a time specification and an if-statement testing the current state together with the guard expression and updating the new state value as well as the action list. See Tables 1 and

Listing 2 The Behavior Model of the ProductionCell System.

```
annex ProductionCell {**
  state variables
    LoadedBlocks :integer;
    StoredBlocks :integer;
  initial
    OutBlockReady!;
    LoadedBlocks := 0;
    StoredBlocks := 0;
  states
    Waiting :initial state;
    Sending, Receiving :state;
  transitions
    Waiting -[(LoadedBlocks < 10) and InBlockReady?, 1]-> Sending
             {InBlockLoaded!;}
    Sending -[InBlockPicked?, 1]-> Waiting
             {LoadedBlocks := LoadedBlocks + 1;}
    Waiting -[OutBlockLoaded?, 1]-> Receiving
             {OutBlockPicked!;}
    Receiving -[true, 1]-> Waiting
               {OutBlockReady!; StoredBlocks := StoredBlocks + 1;}
**};
```

2 for pseudo code samples describing the analysis of an AADL behavior model and the generation of the corresponding TASM main machine, respectively.

In plain English the translation steps can be described as follows:

- For each model, its states are translated into an enumeration type that has the states as its possible values. Moreover, for each model, its states are also translated into a global variable of the enumeration type above. The variable is initialized with the enumeration value corresponding to the model's initial state. For the production cell system of Section 2.1, the generated types and variables is shown in Listing 3 (ProductionCellStateSet is the type and ProductionCellState is the variable).

Listing 3 The AADL behavior annex states of Listing 2 translated into TASM global type and variable.

```
ProductionCellStateSet := {Waiting, Sending, Receiving};
ProductionCellStateSet ProductionCellState := Waiting;
```

- Each state variable (please note the difference between state variables and states as described above) is translated to a global variable with the instance name attached to the variable name in order to avoid name clashes. The variables are initialized with the values given in the initial part of the model. Since a variable must be initialized in TASM, it must also be initialized in the model. Even though TASM supports the possibility to add variables local to a single machine, we do not use that option due to the fact that global variables are visible during simulation, local variables are not, consequently it would become more difficult to simulate the model with local variables. In the production cell system of Section 2.1, the behavior annex of the ProductionCell system has two state variables LoadedBlocks and StoredBlocks, see Listing 2. They are translated into global TASM variables as shown in Listing 4.

Listing 4 The AADL behavior annex state variables of Listing 2 translated into TASM global variables.

```
Integer ProductionCell_LoadedBlocks := 0;
Integer ProductionCell_StoredBlocks := 0;
```

– For each connection between two subcomponent instance ports, a boolean variable representing the connection between the ports is defined. When a signal is sent through the output port, the boolean variable is set to true. The reception of the signal occurs when the boolean variable is read. After it is read, the variable is set to false, so that a new signal can be sent again. In the production cell system of Section 2.1, the behavior annex of the ProductionCell system has several connections, see Listing 2. They are translated into global boolean TASM variables as shown in Listing 5 (in TASM, two slashes introduce a comment). The first boolean variable is initialized to true since the signal is trigged in Listing 2.

Listing 5 The AADL connections of Listing 1 translated into TASM boolean global variables.

```
Boolean Loader_InBlockReady_to_InBlockReady := true;
Boolean InBlockLoaded_to_Loader_InBlockLoaded := false;
Boolean Loader_InBlockPicked_to_InBlockPicked := false;
// ...
```

– For each subcomponent in the AADL system, a main machine with the subcomponent's name is created in the TASM environment. Since the machines are generated from subcomponent instances, one component can be translated into several machines. Then the rules regarding time annotation are applied:
  - The timed feature of the transition is translated into the assignment of the predefined TASM variable *t*.
  - Each transition of the behavior models is translated into a TASM rule in its main machine. The rule is translated into an if-statement with an expression that is a logical conjunction between two subexpressions. The first subexpression tests whether the machine is in the source state, and the second expression is the guard of the transition (omitted if it

consists solely of the value true). The actions of the rule is the action list connected to the transition. However, the first action is the assignment of the state variable to the target state. Moreover, if port signals are received in the guard, their matching boolean variables are set to false at the end of the action list.
- A machine in TASM is active as long as one of its rule is satisfied, otherwise it becomes terminated. In order to keep the machine active even though no rule is satisfied, an else-rule in accordance with Section 2.3 is added.

In the production cell system of Section 2.1, the behavior annex of the ProductionCell system has several transitions, see Listing 2. They are translated into TASM rules as shown in Listing 6. The if-statement transacts the machine state from Waiting to Sending if the number of loaded blocks is less than ten and it receives the InBlockReady signal (see Figure 2. Note that the signal is set to false, so that is can be received again in the future.

Listing 6 The AADL behavior model of the ProductionCell of Listing 2 translated into TASM rules.

```
R1:
{
  t := 1;
  if (ProductionCellState = Waiting) and
     ((ProductionCell_LoadedBlocks < 10) and
     Loader_InBlockReady_to_InBlockReady) then
    ProductionCellState := Sending;
    InBlockLoaded_to_Loader_InBlockLoaded := true;
    Loader_InBlockReady_to_InBlockReady := false;
}

// ...

R5:
{
  t := next;
  else then
    skip;
}
```

The main translation rule is that each AADL subcomponent is translated into a TASM instance. However, there is an additional rule. In many cases, when the subcomponents are translated into TASM machines, it required to establish an environment. Therefore, if a component has subcomponents and a behavior model of its own, it is assumed to hold the system's environment and initialization information, and an instance of the model is translated into a TASM main machine. The two rules fulfill different purposes. The main rule generates the components of the system, and the additional generates its initialization.

Table 1. AnalyseModelParts: Input ⇒ Map × Set × Set × State × Set.

/* Empty */ ⇒ **return** (*emptyMap, emptySet, emptySet, null, transSet*);
featureList name **: type ;** ⇒
(*varMap, initSet, stateSet, initState, transSet*) = AnalyseModelParts(*featureList*);
return (*varMap* ∪ (*name*, **type**), *initSet, stateSet, initState, transSet*);
featureList name **! ;** ⇒
(*varMap, initSet, stateSet, initState, transSet*) = AnalyseModelParts(*features*);
return (*varMap, initSet* ∪ (*name*, **send**), *stateSet, initState, transSet*);
featureList name **: state ;** ⇒
(*varMap, initSet, stateSet, initState, transSet*) = AnalyseModelParts(*features*);
return (*varMap, initSet, stateSet* ∪ *name, initState, transSet*);
featureList name **: initial state ;** ⇒
(*varMap, initSet, stateSet,* _, *transSet*) = AnalyseModelParts(*features*);
return (*varMap, initSet, stateSet* ∪ *name, name, transSet*);
featureList source **-[** *guard, time,*]→ *target* **{** *actionList* **}** ⇒
(*varMap, initSet, stateSet, initState, transSet*) = AnalyseModelParts(*features*);
return (*varMap, initSet, stateSet, initState, transSet* ∪ (*source, guard, time, target, actionList*);

Table 2. GenerateModelParts: Set ⇒ Output.

/* Empty */ ⇒ *generate*("");
transitionSet transition ⇒
GenerateModelParts(*transitionSet*);
(**if** *tr.time* != **null then** *generate*("t := " + *tr.time* + ";");
(*generate*("if (" + *modelName* + "State = " + *tr.source* + ") and (" + *tr.guard* + ") then");
(*generate*(" " + *modelName* + "State := " + tr.target + ";");
(*generate*(" " + *tr.actionList*");
(*generate*(" " + *extractPorts*(*tr.guard*));

3.3 Tool Support

There is a number of tools developed for AADL. One of them is the Open Source AADL Tool Environment (OSATE, aadl.info), which is a plug-in for the Eclipse environment (www.eclipse.org).

AADLtoTASM[2], a contribution of this paper, is an OSATE plug-in that analyzes an AADL model and generates the equivalent TASM specification. It reads the subsystems, ports, and connections of each component as well as its behavior model. The model is parsed in accordance with the grammar of Table 3 with the extension of Section 3.1. Moreover, due to the fact that uninitialized variables are not allowed in TASM, all state variables in the AADL behavior model have to be initialized. In order to properly initialize the TASM variables representing the AADL behavior model states, each model must have exactly one initial state.

AADLtoTASM translates an AADL model extended with time annotations into a TASM model, in order for the model to be simulated in the TASM Toolset Simulation environment. However, the tool does not perform any analysis in addition to the translation.

[2] The tool is available for download, please contact one of the authors

Table 3. The Extended Behavior Annex Grammar.

annex_specification	⇒	*optional_state_variables optional_initialization*
		optional_states optional_transitions
optional_state_variables	⇒	**state variables** *variable_declaration_list* | /* Empty. */
variable_declaration_list	⇒	*variable_declaration* | *variable_declaration_list variable_declaration*
variable_declaration	⇒	*identifier_list* **:** *variable_type* **;**
identifier_list	⇒	**identifier** | *identifier_list* **, identifier**
variable_type	⇒	**{** *identifier_list* **}** | **integer** | **boolean**
optional_initialization	⇒	**initial** *initial_list* | /* Empty. */
initial_list	⇒	*send* **;** | *assignment* **;** | *initial_list send* **;** | *initial_list assignment* **;**
send	⇒	**identifier ! ;**
assignment	⇒	**identifier :=** *expression*
interval	⇒	[**integer_constant , integer_constant**]
optional_states	⇒	**states** *state_list* | /* Empty. */
state_list	⇒	*state* | *state_list state*
state	⇒	*identifier_list* **:** *optional_initial* **state ;**
optional_initial	⇒	**initial** | /* Empty. */
optional_transitions	⇒	**transitions** *transition_list* | /* Empty. */
transition_list	⇒	*transition* | *transition_list transition*
transition	⇒	**identifier -** [*expression* , *time*] →
		identifier *optional_action_list*
time	⇒	**null** | **integer_constant** | *interval*
optional_action_list	⇒	; | **{}** | **{** *action_list* **}**
action_list	⇒	*expression* **;** | *action_list expression* **;**
expression	⇒	*expression binary_operator expression* | **not** *expression*
	|	**(** *expression* **)** | *constant* | *assignment* | *send*
	|	**identifier** | **identifier ?**
binary_operator	⇒	**or and + - * / = != < <= > >=**
constant	⇒	**integer_constant** | **boolean_constant** | **real_constant**
	|	**character_constant** | **string_constant**

4 Simulation

As the ProductionCell system of Section 2.1 has the subcomponents Loader, FeedBelt, BeltToPress, Press, PressToBelt, and DepositBelt, TASM main machines with the same names are generated for each of the subcomponents. Due to the additional rule of Section 3.2, an instance of the ProductionCell is translated into an TASM main machine with the same name. See Listing 3 for the definition of the states and state variables of the production cell system behavior model.

Furthermore, see Listing 2 for the behavior model of the ProductionCell. Extracts of the translated TASM machines are shown in Listings 3, 4, 5, and 6. The system that we want to investigate loads three (the number three is just chosen as an example) blocks into the system and picks three processed blocks from the system. Note, that the signals of input ports are received with the question mark (?) operator and the output ports signals are sent with the exclamation mark (!) operator. Even though it is possible to send data through the ports, only trigger signals are used in this case study. Each transition has a guard as well as a time interval.

In order to evaluate an architecture and find its minimal and maximal time, the production cell system was translated from AADL to TASM by the tool as described above. The minimal and maximal time for each production cell component of the system are defined in Table 4.

Furthermore, the process was simulated one hundred times with random time intervals. Figure 3 describes the results of this simulations. It is also possible to analyze the best and worst case performance by setting the simulator to use the minimum or maximum time, respectively, of each interval. With this method, we could conclude that the minimum time to put three blocks through the system was 72 units and the maximum time was 131 units.

The TASM Toolset does also have the capability to perform a number of different kinds of analysis, such as average time consumption and minimal and maximal time consumption regarding a specific rule. However, due to limited space, we do not describe them in detail in this paper. For further timing analysis of TASM we refer to [14].

Table 4. Transition Time.

System	**Transition**	**Minimum Time**	**Maximum Time**
ProductionCell	Sending	1	1
	Receiving	1	1
	Waiting	1	1
Loader BeltToPress PressToBelt	Magnet On	1	1
	Rotating Arm Forwards	3	6
	Magnet Off	1	1
	Rotating Arm Backward	3	6
FeedBelt DepositBelt	Receiving Block	1	1
	Moving Block	4	8
	Leaving Block	1	1
Press	Move Block to Press Position	1	2
	Pressing	5	10
	Move Block from Press Position	1	2

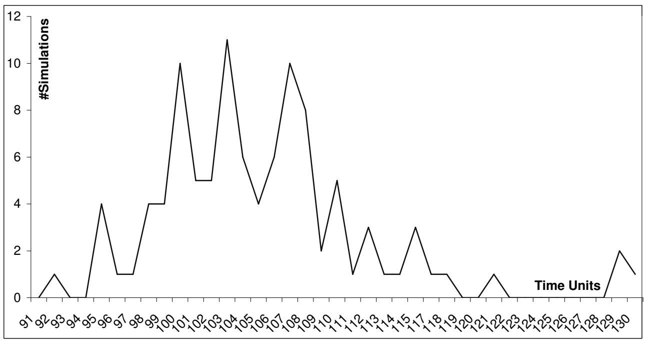

Fig. 3. Simulation of the Generated TASM Model.

The simulation of the case study shows based on the AADL behavior annex extension and the translator AADLtoTASM automatic reasoning about timing behavior of AADL models is possible. As the simulation of the machines are executed in parallel, it would be difficult to perform such simulations manually. Another benefit is that the TASM model can be further translated into a UPPAAL [17] model, where timed model checking can be performed.

5 Related Work

Due to the importance of real-time systems, a considerable number of languages have been formally extended to tackle the problem of verifying real-time requirements and properties. Consequently, in the following we would like to compare the specific contributions of this paper, namely (a) the extension of the AADL notation and its behavior model with timing annotation and (b) the transition of timed AADL models into timed abstract state machines to allow for tool automated simulation of the model with related approaches.

A majority of the related approaches focuses on timed extensions of visual specification formalisms. Known examples are timed Automata [18, 19], Timed Petri Nets [20] and Timed Behavior Trees [21–23]. Timed specification formalisms come with a variety of tools and methods to correctly specify a system and to verify its timing requirements and properties. An example of approaches that help with the correct specification is based on the recently proposed timed patterns [24]. The verification focuses on timed model checking [1] with tools like UPPAAL [17] and KRONOS [25].

Since the use of these model checking tools is also possible for the timed AADL extension as TASM has been chosen as the underlying formal specification formalism in the presented approach. In [26], a formal transition from TASM to timed automata, especially UPPAAL automata, is introduced.

There are also textual notations that have timed extensions, e.g. timed CSP [27], and timed versions of Object-Z [28, 29]. However, the specification of a system in these languages requires expert knowledge in formal methods and practitioners are often discouraged by the strict mathematical formalism. In contrast, the foundation for the approach presented in this paper is with AADL, a well accepted specification language [4]. Furthermore, we argue that the introduced concepts and syntax elements are easy to understand and do not require a massive amount of training for the practitioners that are already familiar with AADL and its behavior model.

Beside the timed extension described in this paper and the use of timed simulations of the underlying TASM model, AADL currently also supports scheduling analysis [30, 31] as a second type of real-time analysis. This scheduling analysis assumes different scheduling strategies (e.g. rate monotonic scheduling) and allows verifying schedulability and end-to-end deadlines. The approach described in this papers is based on a fully concurrent implementation of the architectural elements. Consequently, both approaches are complementary, but we believe that an integration of the two approaches is an interesting topic for future research.

6 Conclusions and Further Work

The main contribution of this paper is an extension of AADL's behavior model to allow for the specification of timed behavior. This extension if based on the formal language of timed abstract state machines (TASM) and consequently techniques like time simulations could be performed to check if an architecture specification meets its real-time requirements and resource constraints.

This paper has furthermore presented a novel tool called AADLtoTASM for transition from AADL with its behavior annex to TASM. OSATE is an Eclipse plug-in and the tool is an OSATE plug-in that reads an AADL file (including its behavior models) and generates the corresponding TASM file.

To evaluate the approach, the production cell case study has been translated from AADL into TASM. This shows that the tool works and that it seems to be valuable when reasoning about AADL models.

There are some features that can be changed in the future. One of them is the stopping criteria for the simulation could be improved. One possible approach could be confidence intervals.

One possible extension of this work is to further translate the TASM model into UPPAAL [17] in order to perform timed model checking. In that case, it would be possible to, in detail, formally define best-case and worst-case time behavior of the AADL model.

Furthermore, an extension to specify probabilistic behavior in AADL's behavior annex, jointly with the specification of the AADL Error Annex [32], would be interesting. As a result also the probabilistic behavior (e.g. probabilistic safety properties [33]) could be analyzed. As an example, Monte Carlo simulation could be performed on these models.

References

1. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for real-time systems. In: Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press (1990) 414–425
2. Grunske, L.: Early quality prediction of component-based systems - A generic framework. Journal of Systems and Software **80** (2007) 678–686
3. Yovine, S.: Model checking timed automata. Lecture Notes in Computer Science **1494** (1998) 114–124
4. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The Architecture Analysis and Design Language (AADL): An Introduction. Technical Report CMU/SEI-2006-TN-011, Society of Automotive Engineers (2006)
5. Feiler, P., Lewis, B.: SAE Architecture Analysis and Design Language (AADL) Annex Volume 1. Technical Report AS5506/1, Society of Automobile Engineers (2006)
6. Ouimet, M., Lundqvist, K.: The TASM Toolset: Specification, Simulation, and Formal Verification of Real-Time Systems. In: Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany. Volume 4590 of Lecture Notes in Computer Science., Springer (2007) 126–130

7. Lynch, N.: Modeling and verification of automated transit systems, using timed automata, invariants, and simulations. Lecture Notes in Computer Science **1066** (1996) 449–459
8. Lewerentz, C., Lindner, T.: Formal development of reactive systems, case study production cell. In Lewerentz, C., Lindner, T., eds.: Formal Development of Reactive Systems. Lecture Notes in Computer Science. Springer-Verlag (1995) 21–54
9. Ouimet, M., Lundqvist, K.: Modeling the Production Cell System in the TASM Language. Technical Report ESL-TIK-00209, Embedded Systems Laboratory, Massachusetts Institute of Technology, Cambridge, MA, 02139, USA (2007)
10. Vestal, S.: Formal verification of the metaH executive using linear hybrid automata. In: Proceedings of the Sixth IEEE Real-Time Technology and Applications Symposium (RTAS '00), Washington - Brussels - Tokyo, IEEE (2000) 134–144
11. Miles, R., Hamilton, K.: Learning UML 2.0. O'Reilly Media (2006)
12. Pilone, D., Pitman, N.: UML 2.0 in a Nutshell. second edition edn. O'Reilly Media (2005)
13. Ouimet, M., Lundqvist, K.: The TASM Language and the Hi-Five Framework: Specification, Validation, and Verification of Embedded Real-Time Systems. In: Asia-Pacific Software Engineering Conference. (2007)
14. Ouimet, M., Lundqvist, K.: The TASM Language Reference Manual, Version 1.1, Massachusetts Institute of Technology, Cambridge, MA, 02139, USA. (2006)
15. França, R.B., Bodeveix, J.P., Filali, M., Rolland, J.F., Chemouil, D., Thomas, D.: The AADL behaviour annex - experiments and roadmap. In: ICECCS, IEEE Computer Society (2007) 377–382
16. Börger, E., Stärk, R.: Abstract State Machines - A Method for High-level System Design and Analysis. Springer-Verlag Berlin And Heidelberg Gmbh and Co. Kg (2003)
17. Behrmann, G., David, A., , Larsen, K.G.: A Tutorial on UPPAAL. In: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT04). Volume 3185., Springer-Verlag (2004)
18. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126** (1994) 183–235
19. Bengtsson, J., Wang, Y.: Timed automata: Semantics, algorithms and tools. In Reisig, W., Rozenberg, G., eds.: Lecture Notes on Concurrency and Petri Nets. Volume 3098 of Lecture Notes in Computer Science. Springer-Verlag (2004) 87–124
20. Winkowski, J.: Processes of Timed Petri Nets. TCS: Theoretical Computer Science **243** (2000)
21. Colvin, R., Grunske, L., Winter, K.: Probabilistic timed behavior trees. In Davies, J., Gibbons, J., eds.: Proceedings of the International Conference on Integrated Formal Methods (IFM 2007). Volume 4591 of Lecture Notes in Computer Science., Springer-Verlag (2007) 156–175
22. Colvin, R., Grunske, L., Winter, K.: Timed behavior trees for failure mode and effects analysis of time-critical systems. Journal of Systems and Software **81** (2008) 2163–2182
23. Grunske, L., Winter, K., Colvin, R.: Timed Behavior Trees and their Application to Verifying Real-time Systems. In: Proceedings of the 18th Australian Conference on Software Engineering (ASWEC 2007), IEEE Computer Society (2007) 211–220
24. Dong, J.S., Hao, P., Qin, S.C., Sun, J., Wang, Y.: Timed patterns: Tcoz to timed automata. In Davies, J., Schulte, W., Barnett, M., eds.: Int. Conference on Formal Engineering Methods (ICFEM'04). Volume 3308 of Lecture Notes in Computer Science., Springer-Verlag (2004) 483–498

25. Daws, C., Olivero, A., Tripakis, S., Yovine, S.: The Tool KRONOS. In: Hybrid Systems III: Verification and Control. Volume 1066 of Lecture Notes in Computer Science., Springer-Verlag (1995) 208–219
26. Ouimet, M., Lundqvist, K.: A Mapping between the Timed Abstract State Machine Language and UPPAAL's Timed Automata. Technical Report ESL-TIK-00212, Embedded Systems Laboratory, Massachusetts Institute of Technology, Cambridge, MA, 02139, USA (2007)
27. Schneider, S.: An operational semantics for timed CSP. Information and Computation **116** (1995) 193–213
28. Dong, J.S., Duke, R., Hao, P.: Integrating object-z with timed automata. In: Int. Conference on Engineering of Complex Computer Systems (ICECCS 2005), IEEE Computer Society (2005) 488– 497
29. Smith, G., Hayes, I.: An introduction to real-time object-z. Formal Aspects of Computing **13** (2002) 128–141
30. Feiler, P.H., Gluch, D.P., Hudak, J.J., Lewis, B.A.: Embedded Systems Architecture Analysis Using SAE AADL. Technical report, CMU/SEI-2004-TN-005 (2004)
31. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Scheduling and memory requirements analysis with AADL. ACM SIGADA Ada Letters **25** (2005) 1–10
32. Feiler, P., Rugina, A.: Dependability modeling with the architecture analysis and design language (AADL). Technical Report CMU/SEI-2007-TN-043, Carnegie Mellon University (2007)
33. Grunske, L., Han, J.: A Comparative Study into Architecture-Based Safety Evaluation Methodologies Using AADL's Error Annex and Failure Propagation Models. In: 11th IEEE High Assurance Systems Engineering Symposium, HASE 2008, IEEE Computer Society (2008) 283–292