ABSTRACT

The Architecture Analysis and Design Language (AADL) has been widely accepted to support the development process of Distributed Real-time and Embedded (DRE) systems and ease the tension of analyzing the systems' non-functional properties. The AADL standard prescribes the dispatching and scheduling semantics for the thread components in the system using natural language. The lack of formal semantics limits the possibility to perform formal verification of AADL specifications. The main contribution of this paper is a mapping from a substantial asynchronous subset of AADL into the TASM language, allowing us to perform resource consumption and schedulability analysis of AADL models. A small case study is presented as a validation of the usefulness of this work.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification—Formal Methods; D.2.11 [Software Engineering]: Software Architectures—Languages

General Terms
Reliability, Verification

Keywords
AADL, TASM, verification, formal methods, formal semantics

1. INTRODUCTION

Distributed Real-time and Embedded (DRE) systems deployed for instance on avionics and aerospace platforms is one of the most safety-critical categories of systems. Usually, DRE systems consist of many local subsystems. Compared with more traditional all-in-one systems, distributed systems tend to have a larger number of non-deterministic aspects. Therefore, designing distributed systems demands more control during the development phases and the use of rigorous methodologies. Moreover, ensuring that the produced system conforms to all stringent functional and non-functional requirements is a very complex and time consuming task. For instance, one common headache with DRE systems is how to, with a high degree of trust, analyze the impact of event triggered aperiodic/sporadic threads to its local subsystem and verify the functional and non-functional requirements of the local system under this circumstance. The model- and component-based development approaches have emerged as attractive options for the development of DRE systems. The Architecture Analysis and Design Language (AADL) [16] has been widely accepted to support the development process of DRE systems and ease the tension of analyzing the systems' non-functional properties. However, the lack of formal semantics limits the possibility to perform formal verification of AADL specifications. Although efforts have been made towards specifying formal semantics of AADL [1, 3, 7–11, 17, 18] there are still some open questions left. For instance, asynchronous interactions, i.e., aperiodic and sporadic threads, are to our knowledge not covered. Within this context, we are motivated to consider an asynchronous subset of AADL in our work of providing a formal semantics of AADL.

We have chosen Timed Abstract State Machine (TASM) [14] as the language to define the formal semantics. TASM is a novel specification language, which has been shown the potential to express formal semantics of AADL [15]. Especially, two distinctive features make TASM stand out. Firstly, TASM supports the specification of both functional and non-functional behavior. The non-functional properties that can be expressed include timing behavior and resource consumption. Secondly, the TASM toolset provides procedures for analysis of completeness, consistency, execution time and resource consumption. Analysis of time-related properties is provided through a translation into timed automata – the input language for the UPPAAL model-checker [2].

The main contribution of this paper is a translation of a chosen subset of AADL into TASM, allowing us to perform resource consumption and schedulability analysis, and schedulability analysis of AADL models. A small case study is presented to show how AADL models can benefit from this work. The rest of the paper is organized using the following structure: Brief overviews of AADL and TASM are presented in Section 2 and Section 3, respectively. Section 4 describes the formal semantics for the chosen subset of AADL. Section 5 presents the corresponding transformation rules. Section 6 shows a case study applying the translation and performing the resource consumption and schedulability
analysis. Some concluding remarks can be found in Section 7.

2. A BRIEF OVERVIEW OF AADL

AADL was released and published as a Society of Automotive Engineers (SAE) Standard AS5506 in 2004 [16]. It is a textual and graphical language, which describes the architecture of component-based systems as an assembly of software components mapped onto components representing the execution platform.

Data, subprograms, threads, threads group and processes collectively represent application software components. Processor, memory, bus and device collectively represent the execution platform. Execution platform components support the scheduling and execution of threads, the storage of data and code, and the communication behavior between processes. Systems are called compositional components. They allow software and execution platform components to be organized into hierarchies with well-defined features. AADL offers an execution model that addresses most of the runtime needs of real-time systems: (1) a set of execution model properties can be attached to each AADL declaration; (2) the semantics of the execution model is also described, namely, the execution semantics of AADL. However, most of it is defined using a natural or semi-formal language. The absence of a precise mathematical semantics makes any pretense of achieving formal verification meaningless [11].

2.1 The Chosen Subset of AADL

The chosen subset includes AADL thread and processor components. AADL thread component is the only component with execution semantics in AADL. In the chosen subset of AADL, an AADL thread can be periodic, aperiodic, or sporadic. Periodic thread dispatches are solely determined by the time interval specified through the Period property value. An aperiodic or sporadic thread dispatch is triggered non-deterministically. But for sporadic threads, a minimum interval time between successive dispatches has to be specified through the Period property value. The property Priority specifies the execution order when more than one threads are ready to execute. The range property Compute_Execution_Time defines the Best Case Execution Time (BCET) and Worst Case Execution Time (WCET). For brevity, we only consider WCET in the paper. AADL processor component is an abstraction of the runtime environment and execution platform, where a scheduler is implicitly included. In this paper, we use the terms scheduler and processor interchangeably. The scheduler plays the role in coordinating all thread executions on one processor as well as concurrent access to shared resources. Various scheduling protocols can be specified according to the Scheduling_Protocol property value. In this paper, we consider a preemptive fixed-priority scheduler.

Definition 1. An AADL-specification $A$ is a pair $<Pr, T>$ where:

- $Pr$ is a processor, which is a triple $<\text{Ident}, T_{\text{Bind}}, \text{Sch}_P>$:
  - $\text{Ident}$ denotes the identifier of the processor, which must be unique in the range of the specification.
  - $T_{\text{Bind}}$ denotes a set of threads bound to the processor, where $T_{\text{Bind}} \subseteq T$.
  - $\text{Sch}_P$ denotes the Scheduling_Protocol property. We assume that the value of the property is "preemptive fixed-priority".

- $T$ is a set of thread components. Let $t$ range over $T$. A thread $t_i$ is a pair $<\text{Id}_i, \text{Sch}_P>$:
  - $\text{Id}_i$ denotes the identifier of the thread $t_i$, which must be unique in the range of the specification.
  - $\text{Sch}_P$ denotes a set of scheduling properties. More specifically, $\text{Sch}_P = \langle \text{Dispatch}_P, \text{Compute}_E, \text{Deadline}, \text{Priority}, \text{Period} \rangle$ of the form Property $i: \text{Ident} \Rightarrow \text{Value}$. We assume that the value of the Dispatch_P can possibly be "aperiodic", "sporadic", and "periodic".

3. A BRIEF OVERVIEW OF TASM

TASM [14] was born at MIT, USA, and now the toolset is being extended at Mälardalen University, Sweden. TASM is a formal language for the specification of embedded real-time systems. The TASM language extends the Abstract State Machine (ASM) [5] to enable the expression of timing and resource consumption.

Definition 2. A TASM specification is a pair $<E, \text{ASM}>$ where:

- $E$ is the environment, which is a triple $<\text{EV}, \text{TU}, \text{ER}>$:
  - EV denotes Environment Variables, the global variables that affect and are updated by machine execution,
  - TU denotes the Type Universe, a set of types that includes real numbers, Integer, Boolean, and user-defined types,
  - ER denotes Environment Resources, a set of named resources. More specifically, ER=$\{(r, s) \mid r$ is the resource name, and $s$ is the resource size$\}$. Examples of resources include memory, power, and bus bandwidth.

- \ASM is the abstract state machine, which is a 4-tuple $<\text{MV}, \text{CV}, \text{IV}, \text{R}>$:
  - MV denotes Monitored Variables, the set of environment variables that affects the machine execution,
  - CV denotes Controlled Variables, the set of environment variables that the machine updates,
  - IV denotes Internal Variables, the set of local variables and they are visible merely inside the machine,
  - R denotes a set of Rules, $R=\langle \text{next}, \text{t}, \text{RR}, r \rangle$:
    - $\text{next}$ is the rule name; $\text{t}$ specifies the duration of a rule execution, which can be a single value or a range value $[t_{\text{min}}, t_{\text{max}}]$ or the keyword next, the next construct essentially states that time should elapse until one of the other rules is enabled. Especially, the lack of a time annotation is assumed
to mean \( t = 0 \); \( RR \) is the resource consumption during the rule execution. Similarly, the omission of a resource consumption annotation is assumed to mean zero resource consumption; \( r \) is a rule of the form "if guard then action", where guard is an expression depending on the monitored or internal variables, and action is a set of updates of the controlled or internal variables. We can also use the rule "else then action" which is enabled merely when no other rules are enabled.}

As an extension of ASM, TASM describes system behaviors as the computing steps of an abstract machine with time and resource annotations. The basic execution semantics of a TASM machine is described as follows: In one step, it reads the monitored variables, selects a rule of which guard is satisfied, consumes the specified resources, and after waiting for the duration of the execution, it applies the update set instantaneously. If more than one rules are enabled at the same time, it non-deterministically selects one to execute. In TASM, time progresses in a fixed constant step called a clock tick which is the minimum time quota. As a specification language, TASM supports the concepts of hierarchical composition, parallelism, communication and synchronization. Hierarchical composition is achieved by means of auxiliary machines - function machines and sub machines. Parallelism is naturally supported, since TASM main machines are executed in parallel. Communication and synchronization between machines can be achieved by communication channels [13] whose semantics is similar to the concept of rendezvous in the Ada programming language.

4. FORMAL SEMANTICS FOR THE SUB-SET OF AADL

In this section, the formal semantics for the chosen subset of AADL is presented in TASM. Firstly, we present the formal semantics for the AADL thread component, which can be regarded as two subcomponents - Dispatcher and Thread. For each sub-component, the formal semantics is described in terms of the sub-component’s states and a corresponding TASM main machine. Secondly, we present the formal semantics for the scheduler in the form of its possible states and a TASM main machine with several Auxiliary Machines (AM).

4.1 AADL Thread

In AADL, periodic, aperiodic, and sporadic threads have the same life cycle [4] but different dispatch protocols. Therefore, we regard the thread component as Dispatcher and Thread.

4.1.1 Dispatcher

As its name implies, Dispatcher represents the behavior of a dispatch protocol which can be periodic, aperiodic, or sporadic according to the Dispatch Protocol property value.

Dispatcher can have two possible states - dispatch (initial state) and wait. The dispatch state denotes that a dispatch of the thread is triggered immediately (if the thread is periodic) or after a non-deterministic time duration (if the thread is aperiodic or sporadic). The wait state denotes that Dispatcher is waiting for the elapse of a specified period to send the next request. In the Dispatcher model, a state variable, disState, is used to denote the current state of the dispatcher. Its initial value is dispatch.

Dispatcher main machine consists of five rules, as shown in Listing 1. Rule Dispatch changes the dispatcher state from dispatch to wait and sends a dispatch request through a global variable disFlag to Thread. We use a variable timer to trace the elapsed time between dispatches. Rule NonDeterministic does nothing, but costs 1 clock tick. When the modeled thread is aperiodic or sporadic, Rule NonDeterministic and Rule Dispatch are always enabled or disabled simultaneously. As a reminder, in TASM, if more than one rules in the same machine are enabled simultaneously, solely one of them will be non-deterministically selected to execute. We introduce this inconsistency purposely to simulate the non-deterministic scenario of dispatching aperiodic and sporadic threads. When the modeled thread is periodic, Rule NonDeterministic is always disabled. Both Rule Waiting and Rule WaitComplete cost 1 clock tick. They are used to simulate the wait state of the dispatcher. For a periodic or sporadic thread’s dispatcher, Rule Waiting is enabled when the period of the corresponding thread does not elapse. On the contrary, Rule WaitComplete is enabled when the period has elapsed and updates its state to dispatch. For an aperiodic thread’s dispatcher, the specified period is zero, so both rules are always disabled. The last rule, Rule Else is used to keep the machine alive, in case no other rules are enabled.

4.1.2 Thread

Thread is responsible for modeling the execution semantics of AADL threads once they are dispatched. Within the AADL context, the complete AADL thread execution model incorporates complex functional and non-functional behaviors. For brevity and simplicity, our model solely focuses on basic functional behaviors - thread dispatching, thread scheduling and execution, but ignores mode transition, remote subprogram, data communication and error recovery. However, these behaviors are subjects for future work.

The possible occurring thread states can be simplified into awaiting_dispatch (initial state), compute, and complete. The awaiting_dispatch state denotes that a thread is awaiting a dispatch request. The compute state denotes a thread is currently computing. The complete state denotes a thread completes its computation and returns to the awaitingDispatch state. More detailed, the compute state can be further refined into two states - ready and running. The ready state denotes that a thread is awaiting the allocation of necessary resources for performing the upcoming execution, such as memory or CPU-time. The running state denotes that a thread is currently occupying the CPU and being executed. In the Thread model, two hierarchical state variables are used - thdState and thdCmpState which respectively describe the current thread state and the current refined compute state. The initial value of thdState is awaiting_dispatch. The initial value of thdCmpState is none which merely denotes the thread is not being executed.

The execution semantics of a thread is expressed as a main machine with six rules, as is shown in Listing 2. Rule WaitDispatch is enabled when Thread is in the awaiting_dispatch state and a dispatch request is received. It changes the state of Thread from awaiting_dispatch to compute, and updates the thdCmpState variable to be the ready state. Rule ComputeReady blocks the Thread machine until a signal through
runThd channel is received from Scheduler that also updates the Thread machine to the running state. A thread within the compute state may be subjected to preemption, where its time and resource consumption must be stalled. TASM does not allow a rule execution to be interrupted by any other rule. In order to model the behavior of preemption, Rule ComputeRunning and ComputeComplete are defined. Both Rule ComputeRunning and ComputeComplete cost 1 clock tick. When the thread is in the running state, Rule ComputeRunning is enabled repeatedly when the amount of elapsed clock ticks is less than WCET-1. Rule ComputeComplete is enabled when the amount of elapsed clock ticks is equal to WCET-1, and then changes thread state to the complete state. Rule Complete is used to complete the current dispatch of the thread. Currently this rule solely changes the thread state back to the awaiting dispatch state, but will be used to implement additional actions of data communication and shared resources in future work. Rule WaitNextDispatch is used to model the idle time between dispatch requests.

4.2 Scheduler

A scheduler grants Thread to execute on the processor based on the specified priority scheme. It ensures that only one thread is being executed on a particular processor. If no thread is in the ready state, the scheduler is idle until at least one thread enters the ready state. A thread will remain in the running state until it completes execution of the dispatch or until a thread with higher priority enters the ready state preempts it. The execution semantics varies according to its scheduling protocol. In this section, we present the execution semantics of a preemptive fixed-priority scheduler.

A scheduler has three possible states - wait_thread (initial state), sche_thread, and run_thread. The wait_thread state denotes that the scheduler is awaiting until a thread enters the ready state. The sche_thread state denotes that the scheduler selects one thread with the highest priority from the set of threads in the ready state. The run_thread state denotes the scheduler grants the selected thread to execute. In the Scheduler model, a state variable designated schState traces the state of Scheduler.

The Scheduler main machine makes use of five auxiliary machines, both sub machines and function machines, as is shown in Table 1. Due to limited space, we do not present them in detail in this paper. The execution semantics of a scheduler is modeled as a main machine with five rules, as shown in Listing 3. Rule WaitThread is enabled when at least one new thread enters the ready state or if there is any thread left in the ready state when the processor is released. Then it updates the scheduler to the sche_thread state. Rule ScheThread is enabled when the scheduler is in the sche_thread state. It selects the thread with the highest priority from the set of threads in the ready state. Rule PreemptThread is enabled if the selected thread has a higher priority than the currently running thread. And the sub machine RUNNEXTTHD() is called to execute. On the contrary, Rule RunThread is enabled if the running thread has a higher priority. This rule changes the Scheduler machine to the wait_thread state. Rule Idle is used to keep the machine alive when no other rules are enabled.

5. TRANSFORMATION TO TASM

Based on the definition of AADL and TASM presented in Section 2 and Section 3 and the formal semantics presented in Section 4, we define two transformation rules for AADL thread and processor component. For the sake of the limitation of pages, we solely show the main part of the rules in Listing 1, 2, 3. The transformation rules are expressed in the form of the LET-IN construction:

- entity =
  LET element1 = body1
  AND element2 = body2 ...
  IN <element1, element2, ...>
  END entity

where the elements between the angle brackets conform to the formal definition of entity.

6. CASE STUDY

In order to illustrate how AADL models can benefit from our formal semantics, we present a case study of the verification of an adapted version of the follower spacecraft guidance system (FSGS) example presented in [6].
AND TASM_Thread(i) =
LET E_thread =
AND ER_thread = power := [POWER_SIZE]; memory := [MEM_SIZE];
IN <EV_thread, TU_thread, ER_thread>
AND ASM_thread =
LET R_thread =
WaitDispatch{ if thdState = waiting_dispatch and disFlag = dispatched then thdState := compute; thdCmpState := ready; disFlag := notdispatched; cmpTime := 0;}
ComputeReady{ if thdState = compute and thdCmpState = ready then runThd();}
ComputeRunning{ t := i; power := POWER_CONSUMPTION; memory := MEM_CONSUMPTION; if thdState = compute and thdCmpState = running and cmpTime < thdWCET, -1 then cmpTime := cmpTime + 1;}
ComputeComplete{ t := i; power := POWER_CONSUMPTION; memory := MEM_CONSUMPTION; if thdState = compute and thdCmpState = running and cmpTime = thdWCET, -1 then thdState := complete; cmpTime := cmpTime + 1; thdCmpState := complete;}
Complete{ if thdState = complete then thdState := awaiting_dispatch; thdCmpState := none;}
WaitNextDispatch{ t := next; else then skip;}
IN <MV_thread, CV_thread, IV_thread, R_thread>
IN E_thread, ASM_thread>
IN TASM_Dispatcher(i) || TASM_Thread(i)
END t,

Listing 2: Transformation Rule of AADL Thread (cont’d from Listing 1)

6.1 Follower Spacecraft Guidance System

FSGS consists of three threads. A sporadic thread (Receiver) receives position data which is sent periodically from the leader spacecraft, updates its own position data, and sends the position data to the Reader thread. A Reader thread reads periodically the position value from the Receiver thread and stores it in a protected object. A Watcher thread "watches and reports" the object to the earth observation station. This model is a typical sub system of a distributed system, with a sporadic thread to exchange data and a set of periodic threads devoted to process data. We assume that all the threads need resources - power and memory, which is shown in Table 2.

6.2 TASM model

The Scheduler machine schedules the execution order of threads based on fixed-priority scheduling protocol. All the threads are hard real-time threads, that is, a missed deadline is regarded as a system failure. The model of the periodic threads Reader and Watcher are respectively expressed by two main machines (Dispatcher and Thread) with the parameters listed above. Although a sporadic thread can theoretically be triggered at any time after a minimum period, we assume a maximum period within which the Receiver thread will be triggered at least once. The maximum period can be the hyper-period of the periodic threads or any other reasonable value. This assumption is reasonable, because as long as the follower spacecraft does not deviate from the leader spacecraft, the FSGS will receive the position data within a maximum period.

6.3 Verification and Validation

6.3.1 Resource Consumption

We use the TASM toolset to analyze resource consumption of the FSGS system. As depicted in Figure 1, the graph shows the aggregate resource consumption in the first period of the FSGS system for each resource - power (upper) and memory (lower), versus the horizontal time axis. Three distinctly high levels represent the resource consumption of the corresponding threads. Because the FSGS system does not contain any parallelism consumption of resources, the minimum and maximum amounts of resources consumed will correspond to the minimum and maximum amounts contained in an individual thread.

6.3.2 Timing Properties

The TASM machines can easily be translated into Timed Automata through the transformation rules defined in [12]. The transformation enables the use of the UPPAAL model checker to verify the schedulability of the FSGS system. In addition, deadlock freedom is an essential property that should be satisfied, which also is a prerequisite for schedulability analysis. Table 3 shows the queries of the properties and the corresponding results.
Table 3: Deadlock Freedom and Schedulability Analysis for FSGS

<table>
<thead>
<tr>
<th>Property</th>
<th>Query</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadlock Freedom</td>
<td>$A[] not deadlock$</td>
<td>Satisfied</td>
<td></td>
</tr>
<tr>
<td>Schedulability</td>
<td>$\forall i : ((\text{Reader.MissDeadline} \lor \text{Watcher.MissDeadline} \lor \text{Receiver.MissDeadline}))$</td>
<td>Satisfied</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Resource Consumption (resources on the Y-axes and time on the X-axes)

7. CONCLUSION AND FUTURE WORK

We present an approach to provide formal resource consumption and schedulability analysis for AADL models of a local subsystem of a DRE system. The approach is to translate the execution semantics of AADL components into rule machines in the TASM language. Periodic, aperiodic and sporadic threads and a preemptive fixed-priority scheduler are covered. We purposely introduce inconsistent rules into the translated TASM machine in order to model the non-deterministic aspects. A small case study is conducted to show how to perform resource consumption and schedulability analysis. Resource consumption analysis is enabled by using the TASM toolset. Schedulability analysis of the translated TASM model is carried out by mapping the TASM model into timed automata.

Future work on this approach will cover a larger subset of AADL components, such as, additional components, the Behavioral Annex, mode change, data communication, the Error Annex, etc. Additional scheduling protocols will be incorporated for analysis and evaluation.

Acknowledgement

This work was partially supported by the Swedish Research Council (VR), and Mälardalen Real-Time Research Centre (MRTC)/Mälardalen University.

8. REFERENCES


