Schedulability Analysis of AADL Models[*]

Oleg Sokolsky and Insup Lee
Department of Computer and Info. Science
University of Pennsylvania
Philadelphia, PA 19104-6389

Duncan Clarke
Fremont Associates
306 Kings Chase
Camden, SC 29020-2160

Abstract

The paper discusses the use of formal methods for the analysis of architectural models expressed in the modeling language AADL. AADL describes the system as a collection of interacting components. The AADL standard prescribes semantics for the thread components and rules of interaction between threads and other components in the system. We present a semantics-preserving translation of AADL models into the real-time process algebra ACSR, allowing us to perform schedulability analysis of AADL models.

1 Introduction

Embedded systems, which now affect most aspects of our everyday lives, are used in a variety of systems from airplanes and cars to medical implants to kitchen appliances. Driven by improvements in hardware technologies and increased end user expectations, embedded systems are becoming increasingly more complex. Enhanced multi-feature functionality and commonplace use of multi-processor and networked platforms makes correct and efficient design of embedded systems a hard task. There is a commonly recognized need for new development frameworks that allow designers to perform efficient exploration of design alternatives and analyze system properties early in the design cycle.

Model-driven development, boosted by the popularity of UML, has become an accepted practice in the design of software-based systems in many application domains. Model-driven development methods are also starting to have an impact on the process of design for multi-processor networked embedded systems. Several proposals for UML profiles for embedded systems have been defined, for example [12, 6, 3]. However, adequate modeling principles for architectural modeling of large embedded systems do not have a universal recognition. An important recent development in this respect is the emergence of AADL. AADL (Architecture Analysis and Design Language) [7] is a new standard for architectural modeling of embedded systems, approved in November 2004 and published as SAE Standard AS5506 [13]. The standard includes textual and graphical versions of the language, and precisely defines execution semantics for its components. This semantic definition provides the basis for analysis of AADL models by off-the-shelf tools.

The AADL standard is complemented by an open-source model development environment OSATE, which supports an XML-based internal representation of AADL models and provides a library of model exploration routines, which operates on the internal representation. This library helps to establish connections with analysis tools.

In this paper, we describe an approach to provide formal analysis of timing properties, including schedulability analysis, of AADL models, which utilizes the AADL semantic definition and builds upon the OSATE framework. In order to analyze a model, we automatically translate it into the real-time process algebra ACSR [9] and use the ACSR-based tool VERSA to explore the state space of the model, looking for violations of timing requirements. The contributions of this work are twofold. On the one hand, we offer a new analysis tool for AADL. The advantage of the tool is that it can handle systems with complex patterns of interaction between components, which in AADL go beyond the scope of more traditional schedulability analysis algorithms. Furthermore, our analysis tool offers a set of failing scenarios in the case when the system is non-schedulable or violations of timing requirements are discovered. By carefully choosing the names in the translated model we make it possible to present failing scenarios in terms of the original AADL model. It is also worth mentioning that in the process of defining the translation, we were validating the carefully defined but informal semantics of AADL.

2 Overview of AADL

Components. The main modeling notion of AADL is a *component*. Components can represent a software application or an execution platform. A component can have a set of externally accessible *features* and an internal implementation that can be changed transparently to the rest of the model as long as the features of the component do not change. Implementation of a component can include interconnected subcomponents.
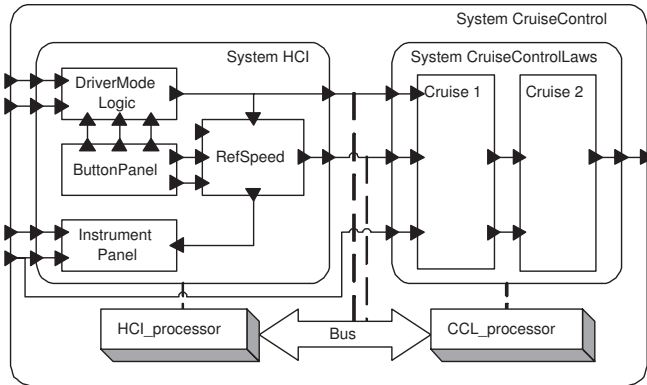
Figure 1. AADL model of a cruise control system

The features of a component include data and event ports and port groups, subroutine call entries, required and provided resources. Interacting components can have their features linked by event, data, and access connections. In addition, application components can be bound to execution platform components to yield a complete system model. Properties, specific to a component type, can be assigned values that describe the system design and can be used to analyze the model. Component types are illustrated in Figure 1. Different component types are shown as different shapes. Solid lines represent connections, while dashed lines represent bindings.

Execution platform components include *processors*, *buses*, *memory blocks*, and *devices*. Properties of these components describe the execution platform. Processors are abstractions of hardware and the operating system. Properties of processors specify, for example, processing speed and the scheduling policy. Buses can represent physical interconnections or protocol layers. Their properties identify the throughput and the latency of data transfers, data formats, etc.

Application components include threads and systems. *Threads* are units of execution. Each thread has an associated *semantic automaton* that describes thread states and conditions on transitions between thread states. A thread can be halted, inactive, or active. An active thread can be waiting for a dispatch, computing, or blocked on resource access; etc. A thread can also be recovering from a fault or in the state of non-recoverable error. Properties of the thread specify computation requirements and deadlines in active states of the thread, dispatch policy, etc. Threads are classified into periodic, aperiodic, sporadic, and background threads. They differ in their dispatch policies and their response to external events. A *system* component is a unit of composition. It can contain application components along with platform components, and specifies bindings between them. Systems can be hierarchically organized.

Figure 1 shows an AADL model of a cruise control system, borrowed from the collection of AADL examples in the OS-ATE release. The system component contains two processors

connected by a bus, and two software subsystems. Each of the subsystems is bound to a separate processor. Threads communicate via data ports. Note how features of a component – in this case, in and out data ports – are mapped by connections to features of its subcomponents.

Connections. Event and data connections between AADL components form *semantic connections*. Each semantic connection has an *ultimate source* and *ultimate destination*. Ultimate sources and destinations can be thread or device components. Starting from an ultimate source, a semantics connection follows connections up the component containment hierarchy via the outgoing ports of enclosing components, includes one "sibling" connection between two components, and then follows connection down the component hierarchy until it reaches the ultimate destination. One of the semantic connections in Figure 1 is between threads RefSpeed in the system HCI and Cruise1 in the system CruiseControlLaws. This connection contains three syntactic connections and is mapped to the bus component. A sporadic or aperiodic thread, which is the ultimate source of an event connection, is dispatched by the arrival of an event via that connection. By contrast, periodic threads are dispatched by a timer and therefore ignore external events, unless they also bring data for processing.

Similarly, semantic access connections describe resources required by a thread that is the ultimate source of an access connection. A resource that serves as the ultimate destination of an access connection is typically a data component. Properties of access connections specify concurrency control protocol for a shared resource.

Modes. AADL can represent multi-modal systems, in which active components and connections between them can change during an execution. Mode changes occur in response to events, which can be raised by the environment of the system or internally by one of the system components. For example, a failure in one of the components can cause a switch to a recovery mode, in which the failed component is inactive and its connections are re-routed to other components. The AADL standard prescribes the rules for activation and deactivation of components during a mode switch. The multimodal nature of AADL models, along with the rich semantics for connections between components makes it difficult to apply standard schedulability analysis algorithms that tend to target restricted task models and communication patterns.

Language annexes. The mechanism of annexes allows users to extend the core language with additional features. For example, an error modeling annex defines additional properties that describe reliability of the system components and a state machine that specifies error states of the system. The use of this annex enables reliability analysis of an AADL model. A behavioral annex allows us to extend thread components with a state machine that specifies computation performed by a thread in more detail. Such a behavioral description refines the default behavior of a thread and enables more precise analysis of the timing behavior of the system.
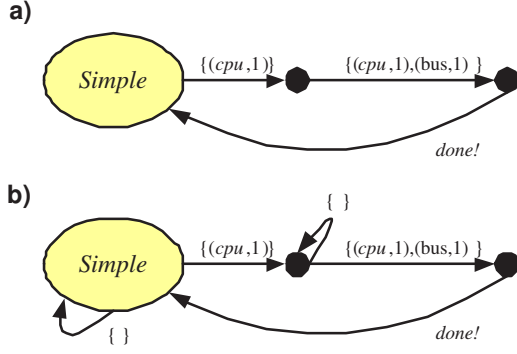
Figure 2. ACSR process with computation and communication steps

3   Overview of ACSR

ACSR [9] is a real-time process algebra that makes the notion of resource explicit in system models. Restrictions on simultaneous access to shared resources are introduced into the operational semantics of ACSR, which allow us to perform analysis of scheduling properties of the system model.

An ACSR model consists of a collection of processes that evolve during the execution of the model. The operational semantics of ACSR defines a transition relation, in which transitions $P_1 \xrightarrow{a} P_2$ describe how process P_1 can evolve into P_2 by performing a step a. Rather than giving a formal description of syntax and semantics, which can be found in several publications [9, 10], we show a pictorial representation for processes. We also use an example that becomes more complex as features of the formalism are introduced.

Computation and communication. ACSR processes can execute two kinds of steps: computation steps and communication steps. Computation steps, which we call here *timed actions*, or simply actions, take time and require access to a set of resources in order to proceed. Access to resources is controlled by priorities that are associated with each resource access. Formally, an action is a set of pairs (r_i, p_i), where p_i is the priority of access to the resource r_i. For an action A, we denote the set of resources to be $\rho(A)$. Communication steps, on the other hand, consist of sending or receiving an instantaneous event. To avoid confusion with event manipulation in AADL models, we will refer to events in ACSR processes as ACSR events. Communication also have priorities associated with them. Figure 2, a shows a simple process that performs a computation step using the processor resource cpu, then performs another computation step that requires, in addition, access to a shared bus represented as the resource bus, and finally announces its completion by sending an event $done$ before restarting.

Resource contention and alternative behaviors. According to ACSR semantics, a timed action cannot be performed if the necessary resources are not available. The process that tries to execute the step will be deadlocked, unless other steps are available in the same state. To allow processes wait for resource access, ACSR models introduce idling steps, which do not consume resources but let the time progress, to allow a process to wait for resources, as shown in Figure 2, b.

Temporal scope of an ACSR process. A process can operate in a *temporal scope* [11], which we represent as a shaded background for the process, as shown in Figure 3. The scope can be exited in one of the three ways: an *exception* represents a voluntary release of control by the process, which is transferred to its exit point, represented pictorially as a white circle; an *interrupt* represents an involuntary release of control, when the control is transferred to a handler process and the activity within the scope is abandoned; the last means of exit is a *time-out*, which occurs a specified duration of time passes since the scope was entered. Scopes can be nested. Pictorial representation may also use labeled arrows to indicate which inputs and outputs can be performed by the process.

Parallel composition and preemption. ACSR processes can be combined in parallel and interact in two ways. Processes can instantaneously send and receive ACSR events. Event communication follows the CCS style of synchronization. The sender and the receiver of matching events take the event step synchronously, performing together an internal step labeled by a special ACSR event τ. For clarity, we also specify the name of the ACSR event that generated the internal step, writing the label as $\tau@name$. Alternatively, a process can perform the step individually, unless the event is *restricted*. Event restriction, therefore forces synchronization of the processes within the scope of the restriction operator. The second means of interaction is implicitly represented by resource conflicts. Processes can perform actions, which take time to execute and require access to a set of resources. Because time progress is global, all processes have to perform action steps together. The following rule for parallel composition specifies that two processes can perform action steps concurrently as long as resources used in each step are disjoint:

$$\text{(Par3)} \quad \frac{P_1 \xrightarrow{A_1} P_1', \; P_2 \xrightarrow{A_2} P_2'}{P_1 \| P_2 \xrightarrow{A_1 \cup A_2} P_1' \| P_2'}, \rho(A_1) \cap \rho(A_2) = \emptyset$$

Access to resources is guarded by priorities, and a process with a higher priority of access can *preempt* the execution of another process. The preemption relation is defined on actions and events. For two actions A_1 and A_2, A_2 preempts A_1, denoted $A_1 \prec A_2$, if every resource used in A_1 is also used in A_2 with greater or equal priority, and at least one resource has a strictly greater priority. As a result of this definition, any resource-using step will preempt an idling step (with an empty set of resources). In addition, an internal step with a non-zero priority will preempt any timed action to ensure progress in the behavior of an ACSR model. The prioritized transition relation for an ACSR process removes preempted transitions from the transition relation.

Figure 3 shows our running example composed in parallel with a driver process, which lets *Simple* complete one iteration. The first action of the driver uses disjoint resources with
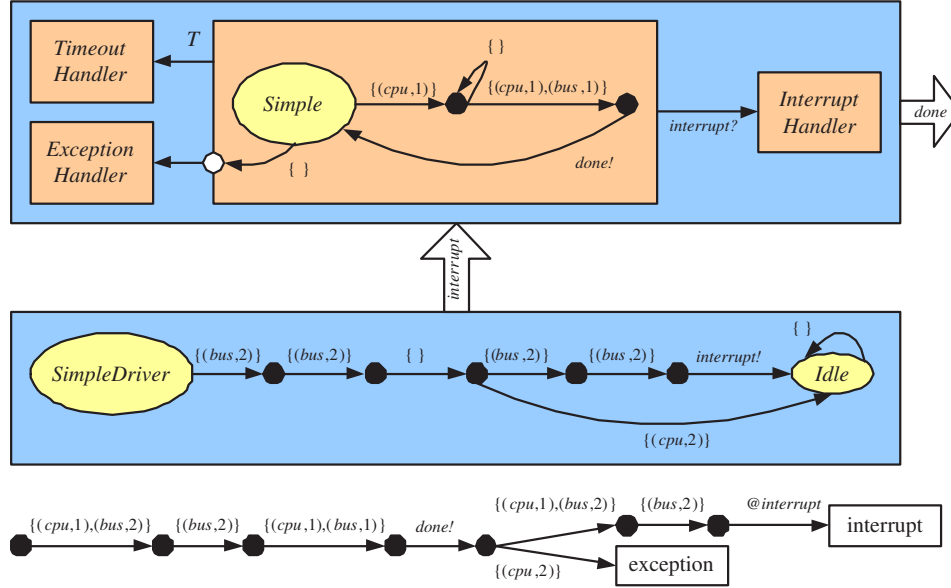
Figure 3. Parallel composition of ACSR processes

the first action of *Simple* and thus they can proceed together. However, the second action uses the same resource bus with a higher priority of access and preempts the execution of *Simple* for one time step. Then, the driver has two alternative behaviors that prevent the process *Simple* from completing the second iteration. One behavior forces an interrupt by sending an ACSR event that synchronizes with the trigger of the interrupt handler. The other behavior preempts *Simple* at the initial state on the second iteration. The alternative idling step takes *Simple* to the exception handler.

Parameterized processes. An ACSR process can be associated with parameters that are changed during an execution of the process. These dynamic parameters are used as variables that keep the history of the execution – for example, the progress of time. At the same time, their use is more restricted than variables (which ACSR processes do not have). Syntactic rules limit the range of each parameter and thus ensure that the parameterized model remains finite-state. The use of parameters in an ACSR process is illustrated in the next section.

4 Translating AADL into ACSR

4.1 Overview of the translation

The main steps of the translation are given by Algorithm 1. Given the limited space, we do not discuss handling of modes in the translation, which is, in general, quite involved. We also omit handling of access connections, which requires encoding of concurrency control protocols. We use the following notation: P is the set of processors in the AADL model; T_p is the set of threads bound to processor p; E_t^{in} (respectively, E_t^{out}) is the set of semantics event or data event connections

that have thread t as the ultimate destination (respectively, ultimate source).

Algorithm 1 AADL to ACSR translation

> **for all** $p \in P$ **do**
>> **for all** $t \in T_p$ **do**
>>> generate a skeleton S_t for t (Section 4.2)
>>> generate an dispatcher D_t for E_t^{in} (Section 4.3)
>>> **for all** $e \in E_t^{out}$ **do**
>>>> populate S_t with events $e!$ (Section 4.4)
>>>> **if** e is mapped to a bus b **then**
>>>>> populate S_t with resource b (Section 4.4)
>>> **for all** $e \in E_t^{in}$ **do**
>>>> generate the queue process for e (Section 4.4)

Continuing our cruise control example in Figure 1, the translation produces six ACSR processes that represent threads and six ACSR processes that represent dispatchers for each thread. All connections in the example are data connections, thus no queue processes are introduced.

Assumptions and restrictions. The translation applies to systems that are *completely instantiated* and *bound*. This means that: 1) The system contains at least one thread and at least one processor components. Each thread has to be bound to a processor; and 2) If the thread is non-periodic (that is, `aperiodic`, `sporadic`, or *background*), each `in event` port and `in event data` port must have an incoming connection. In addition, each thread is required to have the properties `Dispatch_Protocol`, `Compute_Execution_Time`, and `Compute_Deadline` specified. Each processor component that has any threads bound to it must have the property `Scheduling_Protocol` specified.

The current version of the standard AADL assumes that threads in the system are synchronized with respect to a global clock. This assumption matches the timing model of ACSR. In addition, we assume that time is discrete. That is, time is partitioned into fixed-size scheduling quanta and all scheduling decisions are made at quantum boundaries. This assumption means that if a thread is blocked on access to any shared resource, it remains blocked for the remainder of the quantum and any computation performed in this quantum has to be repeated. It also means that access to shared data is modeled as taking the whole quantum, since only one thread can gain access to it during the quantum. As a result of this assumption, analysis will overapproximate timing behavior of a thread and may result in false reports of deadline violations. Precision of the timing analysis can be improved by making scheduling quanta smaller, which tends to increase the size of the state space that needs to be explored. We also assume that the time of data and event delivery across connections in the AADL model is significantly smaller than the scheduling quantum. This assumption allows us to model communication between thread as instantaneous.

4.2 ACSR skeleton of a thread component

Each thread is translated into an ACSR process independently, based on 1) its timing parameters and other properties; 2) its associated connections; and 3) its shared resources. The overall structure of the ACSR process skeleton for a given thread is shown in Figure 4. It directly corresponds to the thread semantic automaton given in the AADL standard. We refer to this process as the *thread skeleton*, because steps within this process can be extended depending on the event and access connections of the thread, scheduling protocol property, etc. Refinements of the skeleton are discussed below. Note that subprocesses of the ACSR thread process can be partitioned into two groups. In one group, the thread is idle waiting for some external event, such as activation or dispatch. In the other group, thread executes some activity, such as activation, deactivation, finalization, or normal computation. These subprocesses relinquish control to the *complete* exit point when the activity is complete, or a timeout occurs when the deadline is reached. The dashed transitions reflect the differences in execution semantics for different kinds of threads. Specifically, recovery is required for sporadic and aperiodic threads when a mode switch occurs during the execution. On the other hand, background threads are dispatched immediately upon initialization. Note the ACSR event *done*, sent when the execution of the subprocess *Compute* completes. That event is received by the dispatcher process of the thread (see below).

An ACSR process representing a thread in the *Compute* state is shown in more detail in Figure 5. The process has two static parameters: minimum c_{min} and maximum c_{max} execution times. They are taken from the property Compute_Execution_Time of the thread component, which gives the range of the execution times. The process is
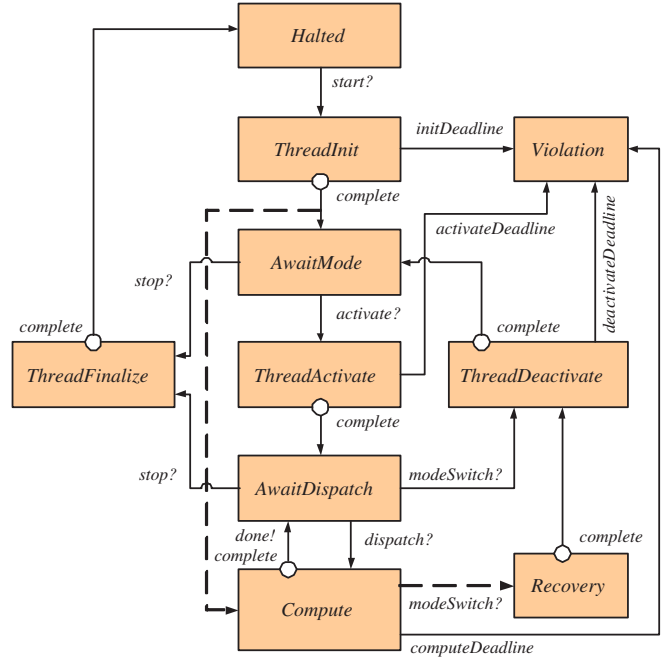

Figure 4. ACSR representation of a single thread component

indexed by two dynamic parameters, e and t. Parameter e represents the amount of execution time that has been accumulated by the thread in the current dispatch. Parameter t represents the total amount of time elapsed since the dispatch.

As the process executes, it performs computation steps that require resource cpu, representing the processor to which the thread is mapped. In addition, an additional set of resources may have to be used, according to the access connections of the thread. This set of resources is generically denoted R. Each computation step increases both dynamic parameters of the process. When the thread is preempted by a high-priority thread, it cannot perform the computation step and moves to the *Preempted* state. There, it performs actions that use resources in R, but not cpu. These steps increase parameter t, but not e. After the number of computation steps exceeds c_{min}, the process can exit its scope via the *complete* exit point and return to the *AwaitDispatch* state. Once the c_{max} has been reached, the process is forced to leave the *Compute*.

As an example of the skeleton refinement, consider again the cruise control model shown in Figure 1. Two of the threads, DriverModeLogic and RefSpeed have outgoing data connections that are mapped to the bus. Thus the bus is modeled as a shared resource that these threads need to access. Output on a data connection is produced as the thread completes its dispatch. Thus the last computation step of the *Compute* state uses both cpu and bus as resources. In all other computation steps in these threads, and all computation steps in the other threads, have $R = \emptyset$ and access only cpu.
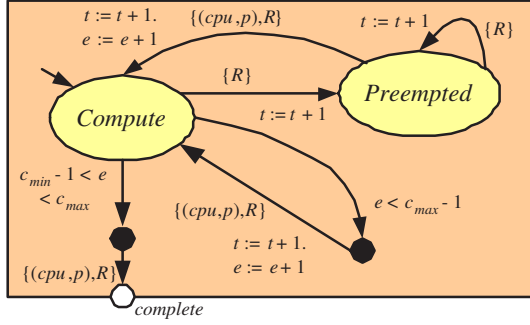
Some of the subprocesses shown in Figure 4 may be ab-

Figure 5. ACSR process for thread computation

Figure 6. Thread dispatchers

sent in the ACSR representation of the task. For example, if the AADL model has only one mode, then subprocesses *ThreadActivate* and *ThreadDeactivate* are not present.

4.3 Tread dispatcher

An AADL thread is dispatched according to its dispatch policy. This policy is captured by the dispatcher process that is generated for each thread in addition to the thread skeleton. The dispatcher sends the *dispatch* event to the thread skeleton that advances the skeleton from the *AwaitDispatch* state to *Compute* state. In addition to thread dispatch, the dispatcher process keeps track of thread deadlines and signals deadline violations by inducing a deadlock into the model execution. Figure 6 shows dispatcher processes for AADL dispatch policies. Figure 6,a shows a dispatcher for a periodic thread. In the initial state, $Dispatcher_p$ sends the dispatch event. Note that the dispatcher cannot idle in this state and has to send this event immediately. Once the event is sent, the dispatcher idles while the thread process is executing. If execution is completed and the ACSR event *done* is received before the timeout d (the deadline of the thread), the dispatched now idles until the next period arrives p time units from the dispatch, and repeats the dispatch cycle. Otherwise, if the deadline timeout happens, the dispatcher process is blocked, inducing a deadlock in the ACSR model that denotes a timing violation.

Aperiodic or background threads are dispatched by an event taken from a queue. The dispatcher process $Dispatcher_a$, shown in Figure 6,b, receives the ACSR event e_deq from the event queue process E_q that corresponds to an incoming event or data event connection of the thread (connection handling is discussed in Section 4.4). When this event is received, the dispatcher sends the dispatch event to the thread skeleton and waits for the ACSR event *done*, which should arrive before the deadline. Note that here, unlike in the case of a periodic thread, the dispatcher can idle waiting for an event to arrive. If there are several incoming event connections, the choice between them may be resolved by assigning priorities to the communication step according to the `Urgency` property of each connection.
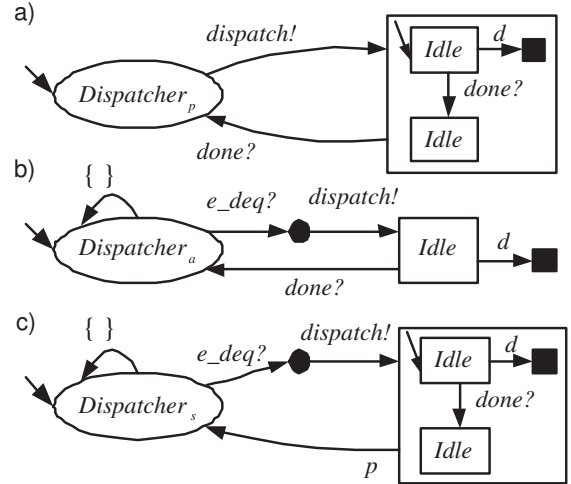
Finally, the dispatcher process $Dispatcher_s$ for a sporadic thread, shown in Figure 6,c, is a combination of the two dispatcher processes discussed above. A dispatch happens when an ACSR event from the queue process is received. However, the next dispatch cannot happen until the minimum separation interval p elapses.

4.4 Events and connections

Sending and receiving AADL events is represented in ACSR by communication steps. Each semantic event or data event connection e in the AADL model is represented by an auxiliary ACSR process E that handles queuing of events at the destination. We introduce two ACSR events: e_q, sent by the source thread and received by E, and e_deq, sent by E and received by the dispatcher for the destination thread.

Sending events. An AADL thread that is the ultimate source of a semantic event or data event connection e can raise an event during its computation. It is represented by a communication step with the output ACSR event with the name e_q. Events can be raised at any time when the thread is executing. The communication step is added as a self-loop to the *Compute* state of the skeleton process in Figure 5. AADL also defines a special completion event that is treated differently. If the implicitly defined out event port `Complete` of the thread is connected, a communication step with the output ACSR event *complete* is performed immediately prior to sending the *done* event.

Since many events can be raised during the execution of a thread, and each such event can cause a dispatch of another thread, analysis results can be very conservative. It can be improved in two ways. On the one hand, if the thread has a behavioral annex associated with it, the state machine in the annex can specify a detailed pattern of event communication. In this case, the state machine of the annex refines the *Compute* subprocess of the thread skeleton. On the other hand,

some reasonable assumptions can be made by the user about the thread behavior. For data event connections in particular, a common behavior of a periodic thread is to send data at the end of its computation period. This is the default treatment of data event connections in our translation.

Queue management. The queue of a connection e is represented by a *counter* ACSR process E that counts up to the number specified by the `Queue_Size` property of the last port of the connection. Queue size of 1 is assumed if the property is not specified. The counter is sufficient for the representation of the queue, since we do not model the attributes of individual events. Therefore, we need to know only the number of events in the queue at any moment during the execution. The counter is incremented by the ACSR input event e_q and decremented by the ACSR output event e_deq, which matches the input event performed by the thread skeleton at dispatch. Property `Overflow_Handling_Protocol` of the port determines the behavior of the queue process when the counter reaches its maximal count. The incoming event can be quietly dropped, introducing a self-loop transition, or can cause an error to occur, in which case it appears as the interrupt of the queue process leading to an error state.

5   Timing analysis

Schedulability analysis of real-time systems modeled in ACSR has been studied in [1]. The schedulability analysis framework of ACSR includes two parts. The first part implements the task model as a collection of concurrent ACSR processes. The translation of the AADL model into ACSR described above accomplishes this part. The second part is the encoding of a scheduling policy as a priority assignment rule.

Any fixed-priority scheduling algorithm, such as rate-monotonic or deadline-monotonic scheduling, can be implemented by considering the set of AADL thread components bound to a particular processor component P in the system and assigning a priority to each thread T_i based on the appropriate properties of the thread. Then, this priority is assigned to every use of the resource that corresponds to P in any timed action of the ACSR thread process for T_i.

Dynamic-priority scheduling can be implemented by using parametric expressions for priorities. For example, in order to reflect the EDF (Earliest-Deadline First) scheduling, we use the following expression as the priority in each access to the processor resource: $\pi_i = d_{max} - (d_i - t)$, where d_i is the deadline of the thread T_i, d_{max} is the largest deadline among the threads assigned to the processor P, and t is the parameter of the ACSR process for thread T_i, which keeps the time since the last dispatch of T_i (see Figure 5). The priority π_i defined above has the property that the earlier the absolute deadline of the current dispatch of T_i, the larger its value. Thus the thread with the earliest deadline will preempt any other thread that is bound to the same processor. Other commonly used dynamic scheduling policies that have encodings in ACSR include the LLF (Least Laxity First) algorithm and

priority-inheritance protocol.

It can be shown that the resulting ACSR model is deadlock-free if and only if every task meets its deadline. Indeed, considering the structure of a thread skeleton in isolation, we can see that it can become deadlocked only if a timeout occurs, that is, if a timing violation is detected or if an error occurs in the case of queue overflow. Further, we can show the absence of blocking due to communication. These two observations, combined, allow us to conclude that the ACSR model that meets its timing constraints is deadlock free. With this, analysis can be performed by state-space exploration of the ACSR process. A deadlock found in the state space of the process indicates a violation of the timing constraints.

The approach to checking thread deadlines by means of an observer process (that is, the dispatcher process of a thread) can be extended to check other timing properties of AADL models. For example, an observer process can capture violations of an end-to-end latency constraint for a data flow from an in port of an AADL system to an output port. Such an observer would be triggered by an input event and, just like a dispatcher process, would deadlock if the output event is not observed by the flow deadline. The caveat is that, if inputs are processes in a pipelined fashion, observer processes need to be spawned dynamically as needed.

Implementation. ACSR models can be analyzed using the VERSA tool [4]. VERSA performs state-space exploration and deadlock detection for a given model. Since the schedulability problem is reduced in ACSR to the problem of deadlock detection, VERSA can be used to perform schedulability analysis. If VERSA finds a deadlock in the model, it reports a trace leading from the start state to the deadlocked state. This trace can be used as the failing scenario, which provides diagnostic information to the user. Schedulability analysis of AADL models has been implemented in the Eclipse framework as a plugin into the OSATE modeling environment. The plugin performs the following three steps. First, the AADL model is translated into the input of the VERSA tool. Second, the model is loaded into VERSA, which looks for deadlocks. If a deadlock is found, the failing scenario is "raised" to the level of the original AADL model. Steps of the trace are reinterpreted in terms of the actions of the components in the AADL model. As a result, we obtain a fully automatic tool for formal schedulability analysis of AADL models that completely hides the specifics of the formal analysis from the user.

6   Related work

Several approaches to formal timing analysis exist that explicitly take scheduling into account. Extensions of timed automata are used in [5, 2, 8], however, continuous time model does not seem to fare well with the discrete nature of scheduling decisions, resulting in much more cumbersome analysis algorithms. More significantly, ACSR is the only formalism that treats resource contention directly in the semantics, avoiding complicated encodings necessary in other formalisms,

which considerably increase the size of the model. Formal schedulability analysis based on state-space exploration of an executable model is related to simulation-based methods offered by tools such as Cheddar [14]. We believe that exploring the state space of a formal executable model offers exhaustive analysis of all possible behaviors, which is very important if there is much uncertainty in the model behavior.

MetaH [15], a precursor to AADL, offers schedulability analysis for rate-monotonic priority assignments. Our approach complements that work by covering other scheduling algorithms as well.

7 Conclusions and future work

We presented an approach to provide formal schedulability analysis for models of embedded systems expressed in an industry-standard language AADL. The approach works by translating the AADL model into an expression in the real-time process algebra ACSR. The basis for the translation is given by the semantic definition for thread components in the AADL standard. Schedulability analysis of the translated ACSR model is enabled by the ability of ACSR to describe resource requirements and use priorities to control access to shared resources. A number of commonly used scheduling disciplines are encoded in this schedulability analysis framework. The schedulability problem is reduced to the problem of deadlock detection, which is performed by the VERSA tool. An important feature of the presented approach is that the diagnostic information produced by VERSA in terms of the translated ACSR model is translated back in terms of the AADL model and can be presented to the user in a convenient time line form.

Future work on this approach includes keeping up with the new developments in the AADL standard. Version 1.1 of the standard is currently being prepared for submission to SAE, and version 2.0 is being planned. Newer versions of the standard will include additional features that will have to be reflect in the translation. One of the new features will be support for hierarchical scheduling, which will require new priority encodings and possibly new component encodings. The other direction of future work is to make the analysis framework more efficient. This involves modifying the translation to produce ACSR models with more compact state spaces as well as improving the state-space exploration efficiency of VERSA.

References

[1] H. Ben-Abdallah, J.-Y. Choi, D. Clarke, Y. S. Kim, I. Lee, and H.-L. Xie. A Process Algebraic Approach to the Schedulability Analysis of Real-Time Systems. *Real-Time Systems*, 15:189–219, 1998.

[2] K. Brink, J. van Katwijk, R. F. L. Spelberg, and W. J. Toetenel. Analyzing schedulability of Astral specifications using extended timed automata. In *Proceedings of Euro-Par '97*, pages 1290–1297, 1997.

[3] R. Chen, M. Sgroi, L. Lavagno, G. Martin, A. Sangiovanni-Vincentelli, and J. Rabaey. Embedded system design using UML and platforms. In *Forum on Specification and Design Languages (FDL '2002)*, Sept. 2002.

[4] D. Clarke, I. Lee, and H.-L. Xie. VERSA: A Tool for the Specification and Analysis of Resource-Bound Real-Time Systems. *Journal of Computer and Software Engineering*, 3(2):185–215, April 1995.

[5] J. C. Corbett. Timing analysis of Ada tasking programs. *IEEE Transactions on Software Engineering*, 22(7):461–483, 1996.

[6] M. Edwards and P. Green. UML for hardware and software object modeling. In *UML for real: design of embedded real-time systems*, pages 127–147. Kluwer Academic Publishers, 2003.

[7] P. Feiler, B. Lewis, and S. Vestal. The SAE AADL standard: A basis for model-based architecture-driven embedded systems engineering. In *Workshop on Model-Driven Embedded Systems*, May 2003.

[8] E. Fersman and W. Yi. A generic approach to schedulability analysis of real time tasks. *Nordic Journal of Computing*, 2005.

[9] I. Lee, P. Brémond-Grégoire, and R. Gerber. A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems. *Proceedings of the IEEE*, pages 158–171, Jan 1994.

[10] I. Lee, J.-Y. Choi, H.-H. Kwak, A. Philippou, and O. Sokolsky. A family of resource-bound real-time process algebras. In *Formal Techniques for Networked and Distributed Systems (FORTE'01)*, Aug. 2001.

[11] I. Lee and V. Gehlot. Language Constructs for Distributed Real-Time Programming. In *Proceedings IEEE Real-Time Systems Symposium*, 1985.

[12] G. Martin, L. Lavagno, and J. Louis-Guerin. Embedded UML: a merger of real-time UML and co-design. In 9th *International Conference on Hardware/Software Co-Design (CODES '01)*, pages 23–28, Apr. 2001.

[13] SAE International. *Architecture Analysis and Design Language (AADL), AS 5506*, Nov. 2004.

[14] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar: a flexible real time scheduling framework. In *SIGAda '04*, pages 1–8, 2004.

[15] S. Vestal. Metah support for real-time multi-processor avionics. In *Proceedings of the Joint Workshop on Parallel and Distributed Real-Time Systems (WP-DRTS/OORTS '97)*, pages 11–21, 1997.