Process-Algebraic Interpretation of AADL Models*

Oleg Sokolsky[1], Insup Lee[1], and Duncan Clarke[2]

[1] Department of Computer and Info. Science, University of Pennsylvania, Philadelphia, PA, U.S.A.
[2] Fremont Associates, Camden, S.C., U.S.A.

Abstract. We present a toolset for the behavioral verification and validation of architectural models of embedded systems expressed in the language AADL. The toolset provides simulation and timing analysis of AADL models. Underlying both tools is a process-algebraic implementation of AADL semantics. The common implementation of the semantics ensures consistency in the analysis results between the tools.

1 Introduction

Distributed real-time embedded (DRE) systems, which once were confined to a few advanced domains such as avionics systems, are now affecting our lives in many ways. DRE systems are used in cars, medical and assisted living devices, in home appliances and factory automation systems. Functionality of such systems if greatly expanded and quality of service requirements remain quite stringent. At the same time, the use of DRE systems in mass-produced systems has unleashed market pressures to compress development time and reduce costs. New development and verification and validation (V&V) methods are needed to produce safe, efficient, and competitive DRE systems.

Early evaluation of the system design is important for successful and timely development. Correction of errors in the design becomes progressively more expensive later in the design process. Of course, many details of the system are not know in the early stages of development, and only high-level evaluation is possible. Architectural modeling offers a structured way to collect available in information about the system and incrementally refine it as the design progresses. An architectural model also allows developers to apply high-level analysis techniques that can quickly uncover problems. Since architecture-level analysis tend to be approximate and efficient, design-space exploration can be performed by varying different aspects of an architectural model and comparing outcomes of analysis for different architecture variants.

AADL [6, 13] is a standard for the architectural modeling of DRE systems. It allows developers to describe a system as a collection of interacting components

* This research has been supported in part by grants AFOSR STTR AF04-T023, NSF STTR IIP-0712298, NSF CNS-0720703, and AFOSR FA9550-07-1-0216.

and connections between them, abstracting away the functionality of components that is not precisely known at early stages of system development. The standard defines interchangeable textual and graphical modeling notations and gives precise, if mostly informal, semantics for the components and connections. AADL modeling is supported by an open-source development environment OSATE, which provides an extension API for the development of analysis plugins that operate on the OSATE internal representation of AADL models.

In this paper, we describe two analysis techniques for AADL models and present their implementation in the Furness toolset, implemented as an OSATE plugin. One technique is an AADL simulator that allows the user to visually follow the high-level execution of the system and track resource utilization. The other technique is schedulability analysis that determines whether the system has enough resources to satisfy the timing constraints. Both analysis technique, along with many other analysis techniques developed for AADL, rely on AADL semantics. We argue that tools that implement these analysis techniques need a common interpretation engine to ensure that all tools treat AADL semantics consistently. Furness toolset uses an encoding of AADL semantics in a real-time process algebra ACSR [9], which provides a common semantic foundation for for different analysis tool in the toolset.

The paper is organized as follows. Section 2 presents an overview of AADL and its behavioral semantics. Section 3 presents the real-time process algebra ACSR and formal schedulability analysis. In Section 4 we turn to the architecture of the Furness toolset and present the translation of an AADL model into ACSR. Finally, Section 5 concludes with a discussion.

2 Introduction to 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. 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 used to analyze the model. Main 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. A thread can be halted, inactive, or active. An active thread can be waiting for a dispatch, computing, or blocked on resource access; etc. We discuss thread semantics in more detail below. Properties of the thread specify computation requirements and deadlines in active states of the thread, dispatch protocol, etc. Threads are classified into periodic, aperiodic, sporadic, and background threads. They differ in their dispatch protocol 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.

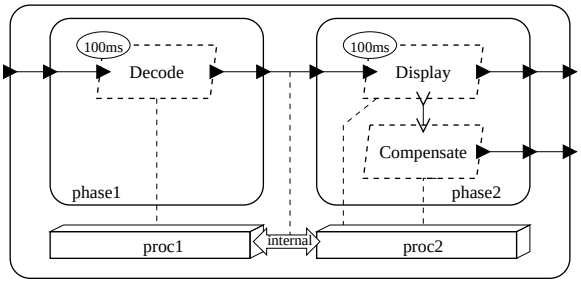

Fig. 1. A simple AADL model

Figure 1 shows a simple AADL model of a stream processing system. The system component contains two processors connected by a bus, and two software subsystems, one of which contains a single periodic thread and the other one contains two threads, one periodic, the other aperiodic. Each of the subsystems is bound to a separate processor, while the connection between threads bound to different processors is bound to the bus. Threads communicate via data or event ports, denoted by filled and blank angles, respectively. Features of a component are mapped by connections to features of its subcomponents.

Semantics of AADL threads. The AADL standard specifies semantics for each AADL component, most of which are precise and detailed but informal. An exception is the thread component. Semantics of a thread are formalized using a hierarchical *stopwatch automaton* that describes thread states and conditions on transitions between thread states. Figure 2 shows the main part of the thread automaton, omitting initialization, error recovery, and mode switching. The automaton uses two clocks, t and c, which represent elapsed time and accumulated execution time, respectively. First derivatives of the clock functions are denoted δt and δc. Elapsed time always evolves at the same speed ($\delta t = 1$). When the thread is not executing – for example, preempted by another thread – the clock c is stopped ($\delta c = 0$). The invariant of the suspended state and the predicate

$Enabled(t)$ depend on the dispatch protocol property. For example, for a periodic thread, the invariant is $t \leq Period$ and $Enabled(t)$ is $t = Period$.

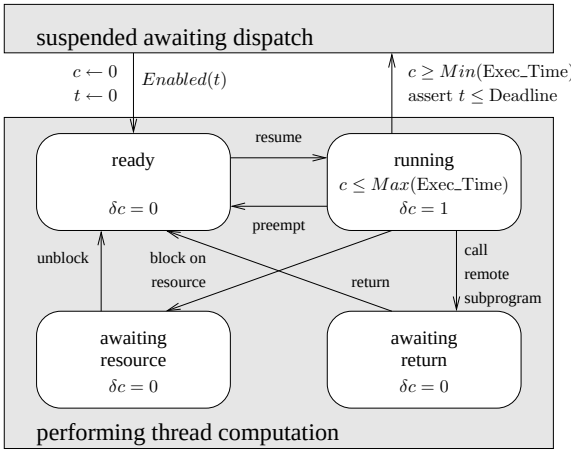

Fig. 2. Semantic automaton for an AADL thread

Connections. Event and data connections between AADL components form *semantic connections*. Each semantic connection has an *ultimate source* and *ultimate destination*, which can be thread or device components. Starting from an ultimate source, a semantic 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 connections down the hierarchy to the ultimate destination. One semantic connection in Figure 1 is between threads `Decode` and `Display`. This data connection contains three syntactic connections and is mapped to the bus component. A semantic event connection exists between threads `Display` and `Compensate`. `Compensate`, being an aperiodic thread, is dispatched by the arrival of each event via that connection. By contrast, periodic threads are dispatched by a timer. 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. Currently, Furness toolset does not support multiple modes and we do not discuss this aspect of AADL any further.

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.

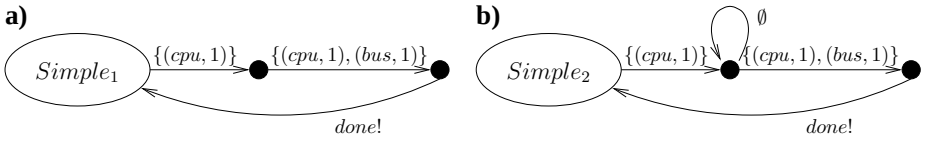

Fig. 3. ACSR process with computation and communication steps

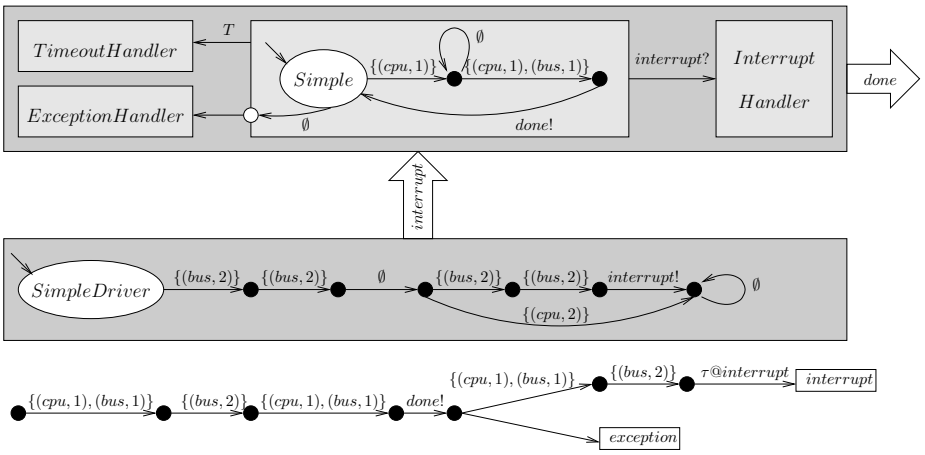

Fig. 4. Parallel composition of ACSR processes

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 3, 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 alternative steps are available. 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 3, b.

Temporal scope. A process can operate in a *temporal scope*, which we represent as a shaded background for the process, as shown in Figure 4. 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 *timeout*, which occurs a specified duration of time passes since the scope was entered.

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 4 places our running example into a temporal scope that composed in parallel with a driver process, which lets *Simple* complete one iteration. The first action of the driver uses disjoint resources with 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 synchronizing 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. 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.

Tool support. Modeling and analysis of real-time systems using the ACSR formalism is supported by the tool VERSA [5]. Originally designed as a rewrite engine for ACSR terms with respect to the strong prioritized bisimulation [9], VERSA is primarily used as a state-space exploration and reachability analysis tool. VERSA uses efficient explicit-state representation of the state space, identifying each state with a normalized ACSR term, extended with a timeout value for each temporal scope operator. Because of explicit state representation, construction of the state space takes time at tool startup, however state transitions take constant time, making VERSA an efficient simulator.

3.1 Schedulability analysis with ACSR

We adopt the schedulability analysis approach described in [3]. In this approach, a real-time system that consists of a collection of tasks is modeled by a parallel composition of ACSR processes constructed in the following way. Each task is represented as an ACSR process that captures task states – such as inactive, ready, running, preempted, etc. – and reflects dependencies on other tasks in the system. In addition, a separate ACSR process models the task *dispatcher*. The dispatcher models the pattern of task dispatches, either by a clock or by

incoming events. The scheduler of the real-time system is not represented explicitly. Instead, it is encoded in the priorities of actions that access the processor resource in task models. For fixed-priority scheduling, the value of thread priority is used. To encode dynamic-priority schedulers, parametric expressions are used as priorities. We give a concrete example of such parametric expressions in Section 4.1. As shown in [3], the resulting ACSR model is deadlock-free if and only if the respective collection of tasks is schedulable by the specified scheduler. Thus, schedulability analysis is reduced to deadlock detection.

4 The Furness Toolset

The Furness toolset provides behavioral analysis of AADL models using VERSA as a state-space exploration engine. The overall architecture of the tool is shown in Figure 5. In the figure, tools from the underlying development framework are shown shaded, while modules that comprise the Furness toolset are white. Furness is a plugin for the OSATE development environment for AADL, which is in turn a plugin into the popular open-source Eclipse framework. The tool operates on AADL instance models, which are created by OSATE from declarative AADL models. When Furness is invoked, the translation module produces an ACSR model from the AADL instance model, which is given as input to the VERSA tool. VERSA processes the generated model and builds its state space. At this point, Furness is ready to perform analysis.

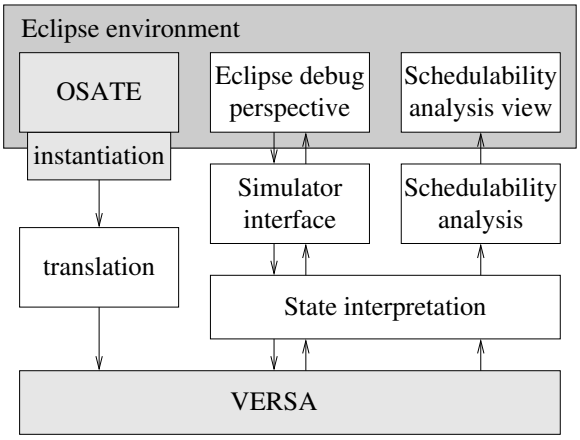

Fig. 5. Furness toolset architecture

An important requirement in the design of the tool was that the user needs to be unaware of the underlying VERSA implementation of the AADL semantics. By hiding VERSA, we achieve to desirable goals. On the one hand, we will be able to employ a different implementation of AADL semantics and make the

switch transparent to the end user. More importantly, the user will be spared the details of the ACSR formalism. The target user of the Furness toolset is an engineer, who is unlikely to be well versed in formal methods. Shielding the user from the formal details will help the adoption of the tool. To achieve this, we introduced the state interpretation module, an abstraction layer over the ACSR model that transforms ACSR execution traces into AADL-level traces.

Both simulation and timing analysis involve state space exploration of the ACSR model. The state interpretation module maintains the correspondence between states of the ACSR model and states of the AADL model. When a transition in the ACSR model is taken, the state interpretation model identifies whether the transition corresponds to an AADL-level event (such as tread dispatch or completion) and updates the AADL model state. Note that multiple ACSR transitions may correspond to a single AADL-level step. In this case, chains of transitions are collapsed by the module into a single step.

The presentation layer of the Furness toolset consists of the standard Eclipse debug perspective, which is used as the user interface for the simulator, and several custom views that present timing analysis result and show the unfolding simulation trace. The simulator interface module handles user requests and manages the simulation state such as breakpoint status. It converts user requests into state interpretation commands and passes the outcome of state interpretation to the user interface, maintaining a bi-directional communication with the state interpretation module. By contrast, timing analysis displays only the results produced by VERSA, resulting in a one-way interaction.

Translation of AADL into ACSR and schedulability analysis have been described in [14]. We reproduce some of this description here for the sake of completeness, concentrating primarily on examples that illustrate the translation.

4.1 Translation of AADL into ACSR

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 specify properties `Dispatch_Protocol`, `Compute_Execution_Time`, and `Compute_Deadline`. 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 discrete global clock. These assumptions match the timing model of ACSR. 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 threads as instantaneous.

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. 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. The overall structure of the thread skeleton, excluding initialization and mode switching parts, is shown in Figure 6. It directly corresponds to the thread semantic automaton given in Figure 2. Refinements of the skeleton are discussed below, when we consider event and data connections. The skeleton 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 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. 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 takes the alternative that leads to the *Preempted* state. There, it performs idling steps, which 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 scope an return to *AwaitDispatch*.

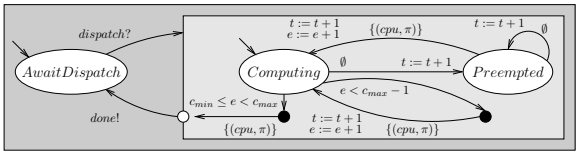

Fig. 6. ACSR process for thread computation

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 7 shows dispatcher processes for AADL dispatch policies. Figure 7,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, ensuring that dispatches happen precisely every p time units. 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 dispatcher idles

until the next period and repeats the dispatch cycle. Otherwise, the deadline timeout happens and the dispatcher process is blocked, inducing a deadlock in the ACSR model that denotes a timing violation.

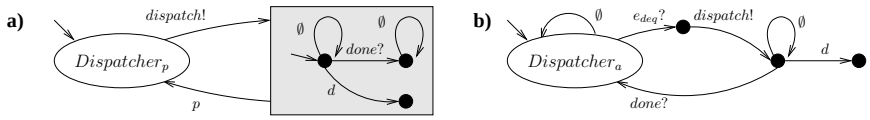

Fig. 7. Thread dispatchers

Aperiodic threads are dispatched by events taken from a queue. The dispatcher process $Dispatcher_a$, shown in Figure 7,b, receives the ACSR event e_deq from the event queue process E_q that corresponds to an incoming event connection of the thread (see below). 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 the dispatcher can idle waiting for an event to arrive. The dispatcher process for a sporadic thread 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.

Event 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. The process E implements a *counter*, which 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 size of the queue and overflow handling logic are obtained from the properties of the port feature.

An AADL thread that is the ultimate source of a semantic event or data event connection e can raise an event during its computation. We refine the skeleton of this thread with a communication step with the output ACSR event e_q, added as a self-loop to the *Computing* state of the skeleton process in Figure 6.

Data connections. Data connections in AADL model sampled communication and thus do not require queues. However, AADL introduces the notion of an *immediate* connection that has the following semantics. Whenever the two threads that are the source and the target of an immediate data connection are dispatched logically simultaneously, the execution of the target thread is delayed until the source thread completes its execution and the data is made available to the source thread. In order to implement the prescribed semantics, first refine the dispatcher of the target thread to accept a special event $block_d$ before activating the thread, and the skeleton of the source thread is refined with an

auxiliary event to announce its completion. Then, we introduce an auxiliary ACSR process that interacts with the dispatchers of the two threads to detect simultaneous dispatches and, if so, delays sending $block_d$ until the source thread completes. Otherwise, $block_d$ is offered immediately.

4.2 Trace abstraction

During the analysis, VERSA maintains the current state of the model. Interacting with VERSA, the state interpretation module observes execution traces of the ACSR model, but not the state directly. To help identify state changes through trace steps, we add to the ACSR model transitions that do not directly correspond to any activity on the AADL level. We call such transitions *bookkeeping steps*. For example, an ACSR timed step specifies what resources were used in the step, but not which process was using the resource. We thus had to introduce a bookkeeping step that occurs immediately after the time step and whose event identifies the thread that used the resource. Internal synchronization between processes in the AADL model – for example, between the thread process and an auxiliary process that implements a data connection – also introduce internal bookkeeping steps.

The state interpretation module has to abstract away bookkeeping steps. A single AADL-level step reported by the module corresponds to a sequence of bookkeeping steps followed by a relevant step. This means that, in the context of a simulator, a single step request from the user results in multiple calls to VERSA until the right step is found. When multiple alternative steps are possible from the current state, the module has to figure out, which selections need to be made. To do this, when the state of the model changes, the module preprocesses available alternatives, converting them into tuples of numbers that encode selections. Consider the following example. Let the current state of the model be represented by the ACSR term $\tau.\text{dispatch\_}t_1 + \tau.(\text{raise\_}e_1 + \text{receive\_}e_2)$. The term has three alternative AADL-level steps, first represents dispatch of a thread, the second one model event handling. Before presenting the choices to the user, the module internally represents them as tuples $(1,1)$, $(2,1)$, and $(2,2)$, respectively. Then, if the user requests to raise event e_1, the module will request two steps from VERSA, selecting the second alternative for the first one and the first alternative for the second step.

4.3 Timing analysis

Timing analysis in the Furness toolset combines two techniques: schedulability analysis and response time calculation. To present results to the user, we introduce two new Eclipse views. One for presenting a failing scenario when the AADL model is not schedulable; the other is for displaying thread response times.

We use the deadlock detection capability of VERSA to perform schedulability analysis. If a deadlocked state is found, VERSA produces a counterexample in the form of an execution trace that leads to the deadlocked state. This trace is lifted to the AADL level and is presented to the user as a failing scenario

using the schedulability analysis view. Representation of the failing scenario is similar to the simulation trace, which is discussed in the next section. We can, in fact, use the simulation engine to let the user replay the scenario - however, this feature is currently not implemented.

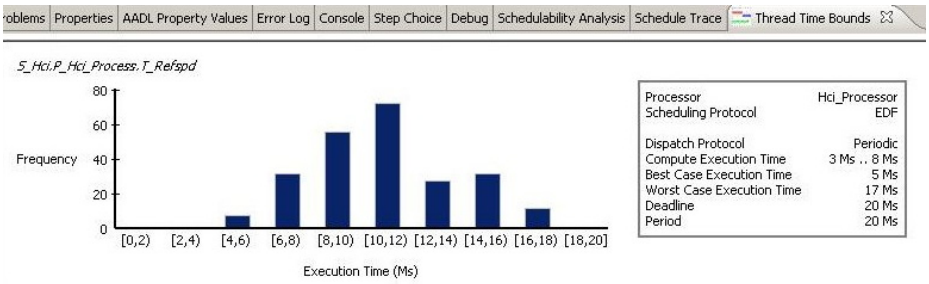

Fig. 8. Response time analysis report

Furness toolset also performs response analysis for a schedulable task set and presents results graphically to the user. For each execution path in the system, response time of every thread is calculated. Then, a histogram is constructed showing the number of execution paths that exhibit a given response time. The histograms are presented to the user through the time bounds view. An example is shown in Figure 8. Intuitively, a thread that has response times along the majority of its execution paths close to its deadline may not be robust enough and can be the target for improvement. This analysis implicitly assumes that all execution paths are equally likely. One can capture the likelihood of different paths to come up with a more precise characterization of thread robustness. For example, PACSR [12], a probabilistic extension of ACSR, can be used for this purpose. However, the necessary information, such as probability distributions of thread execution times and event arrivals, cannot be extracted from AADL without introducing new properties into the model.

4.4 AADL simulation

The Furness simulator is based on the Eclipse debug perspective, familiar to anyone who has used Eclipse to develop code. The perspective provides standard controls to start, pause, resume, and stop simulation, as well as views to display variable values during simulation and manage breakpoints. We also created a custom view that shows the execution trace of the model up to the current state. The view is shown in Figure 9. At left, threads involved in the simulation are listed, grouped by the processor they are bound to. To concentrate on a particular subsystem, the user can hide threads of a selected processor. Each thread and each processor has a line in the trace, which shows its color-coded state at every time instance. A thread is shown as running, blocked waiting for input/preempted, or inactive. A processor is shown as idle or busy.

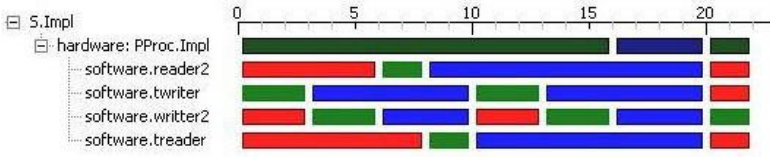

Fig. 9. Execution trace view

The simulator offers two modes, interactive and continuous. In the interactive mode, the user manually requests the execution of a step, which can be either a micro-step or a macro-step. A micro-step corresponds to a single change in the state of the AADL model, that is, a thread being dispatched or completed, an event being raised or delivered to a queue, etc. Multiple micro-steps can occur simultaneously and observing them individually can be tedious. A macro-step, then, is a sequence of micro-steps, followed by a time-consuming step.

When a micro-step is invoked, the user interface layer invokes the simulator interface module that, in turn, passes it to the state interpretation module. It interacts with VERSA and processes its output and returns the new available steps to the user. Macro-steps are performed by internally setting a breakpoint at the next time step and switches to the continuous mode.

In the continuous mode, the simulator keeps executing steps until a deadlock in the model is reached or some pre-defined condition, such as a breakpoint, occurs. Currently, the simulator offers only time breakpoints, when an execution is stopped after a fixed number of time steps. Other kinds of breakpoints, for example upon raising a specific event, can be easily added through the breakpoint interface of the Eclipse debug perspective. An important option for the continuous mode is the resolution of alternative steps. Alternatives can be resolved randomly, or the execution can be paused to let the user select the alternatives.

5 Discussion and Conclusions

We have presented a representation of AADL semantics using a real-time process algebra. This semantic representation is used as a common foundation for an AADL simulator and a schedulability analysis tool. It also can be used by any other tool that requires exploration of the state space of the AADL model. The semantic representation is based on the hybrid automaton describing states of thread components and utilizes a number of relevant properties of thread and processor components and semantic connections in the model.

The semantic representation presented in this paper reflects the fragment of AADL supported by the Furness toolset. Many of the restrictions can be lifted in the future. Most notably, the assumption that communication is instantaneous is unrealistic when connections are mapped to buses. By treating buses as resources in the system and incorporating bus scheduling protocols into the translation, the semantic representation can be made more realistic.

Other formalisms can be used to create a similar semantic representation for AADL. In [4], the authors describe a translation of AADL into BIP [2]. Petri nets are used to capture the semantics of AADL in the Ocarina toolset [8] and also in [11]. Linear hybrid automata in the TIMES tool [1] are used in [7] to provide simulation of AADL threads without execution time uncertainty. We believe that ACSR is a more suitable semantic representation, since is it incorporates the notion of a resource directly in the formalism. Resources in the generated model correspond to platform components, making the translation more direct.

References

1. T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES - a tool for modelling and implementation of embedded systems. In *Proceedings of TACAS '02*, pages 460–464, Apr. 2002.
2. A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time systems in BIP. In 4^{th} *IEEE International Conference on Software Engineering and Formal Methods (SEFM06)*, pages 3–12, Sept. 2006.
3. 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.
4. M. Chkouri, A. Robert, M. Bozga, and J. Sifakis. Translating AADL into BIP – application to the verification of real time systems. In *Workshop on Model Based Architecting and Construction of Embedded Systems*, pages 39–54, Sept. 2008.
5. 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, Apr. 1995.
6. 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.
7. S. Gui, L. Luo, Y. Li, and L. Wang. Formal schedulability analysis and simulation for AADL. In 2^{nd} *International Conference on Embedded Software and Systems*, pages 429–435, July 2008.
8. J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite. *ACM Transactions in Embedded Computing Systems (TECS)*, 7(4), July 2008.
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, A. Philippou, and O. Sokolsky. Resources in process algebra. *Journal of Logic and Algebraic Programming*, 72:98–122, May/June 2007.
11. D. Monteverde, A. Olivero, S. Yovine, and V. Braberman. VTS based specification and verification of behavioral properties of AADL models. In *Workshop on Model Based Architecting and Construction of Embedded Systems*, Sept. 2008.
12. A. Philippou, R. Cleaveland, I. Lee, S. Smolka, and O. Sokolsky. Probabilistic resource failure in real-time process algebra. In *Proceedings of CONCUR*, 1998.
13. SAE International. *Architecture Analysis and Design Language (AADL), AS 5506*, Nov. 2004.
14. O. Sokolsky, I. Lee, and D. Clarke. Schedulability analysis of AADL models. In *Workshop on Parallel and Distributed Real-Time Systems*, Apr. 2006.