AADS+: AADL Simulation including the Behavioral Annex

Roberto Varona-Gómez, Eugenio Villar  
University of Cantabria, Av. Los Castros s/n, 39005 Santander, Spain  
{roberto, villar}@teisa.unican.es

Abstract

AADL has been proposed for designing and analyzing SW and HW architectures for real-time mission-critical embedded systems. However, it does not support the expression of the behavior of a system in detail, so a behavioral annex has been defined. In this paper we propose AADS+, an AADL simulation tool that supports the performance analysis of the AADL specification, enriched with behavior specifications, throughout the refinement process from the initial system architecture until the complete, detailed application and execution platform are developed. In this way, AADS+ enables the verification of the initial timing constraints during the complete design process.

1. Introduction

AADL [1-3] was developed as a standard of the SAE to enable the description of task and communication architectures of real-time, embedded, fault-tolerant, secure, safety-critical, SW-intensive systems. It is used to describe the software and hardware components of a system and the interfaces between them. However, AADL does not support the expression of behavior in detail. At most, it is possible to specify the non-deterministic behavior of a thread as a set of subprogram calls, and application behavior relies mainly on source code written in source languages. The behavioral annex [4] has introduced high-level composition concepts and a richer state representation than the standard AADL mode automata. The behavior is specified using extended automata that may trigger a transition by an event, a Boolean expression, etc. A transition may trigger one or more actions such as assignment of values to variables, sending data, events, etc. The annex mainly declares states and transitions with guards and an action part. Guards and actions can access ports and data subcomponents declared in the AADL component to which they are attached.

There is a commonly recognized need for new development frameworks that enable designers to perform efficient exploration of design alternatives and analyze system properties throughout the design cycle. Some system properties can be obtained by static analysis. Many other properties can only be obtained through simulation. In any case, system simulation is necessary for performance analysis under real execution conditions. System simulation validates the correct dimensioning of the system, detection of locks, missed deadlines and other potential problems caused by the complex interaction among components that can be found in a real system. The earlier all those problems are detected, the less the associated cost of correcting them [5].

Evolutionary prototyping is becoming a well-accepted development approach in Model-Driven Engineering (MDE) [6]. The design flow is based on a central model that is refined unless it is satisfactory. Programs can be generated from this model and constitute intermediate versions of the product. The last refined model corresponds to the final system. A prototyping-based design process is of interest to verify as early as possible, the impact of deployment decisions, or the use of a particular HW/SW component in the system.

In this paper, a complete AADL simulation methodology including the behavioral annex is presented. This methodology has been implemented in the tool AADS [7]. AADS is a simulation framework that can support prototype-based design allowing the functional and non-functional (execution times, power consumption, etc.) verification of the system while it is being refined right through to the final implementation. AADS is based on SystemC that has become the standard language for modeling and simulation of HW/SW embedded systems [8]. The SystemC framework supports the seamless integration of any HW component and an easy optimization of the executive platform.

---

4 This work has been partially supported by the Spanish MICyT through the ITEA 05015 SPICES Project and the TEC2008-04107 project.
The contents of the paper are as follows. The following section analyzes the state of the art. In Section 3, the previous work carried out with AADS is summarized. Then, the SystemC model generation methodology from AADL behavioral annex is explained. Next, a case study is presented and finally conclusions are stated.

2. State of the art

Several authors have considered the behavioral annex in their research on AADL. Some of their papers were written in the initial stage of the behavioral annex so they were intended to evaluate, promote and disseminate it. P. Dissaux et al. [9] present a proposal for a behavioral annex to the AADL standard. They explain how to implement the behavioral annex with the Stood tool, a graphical AADL editor that can import and export AADL textual specifications. R. Bedin et al. [10] evaluate the behavioral annex through a flight software design in the ArchiDyn project. They require new synchronization primitives for AADL runtime and support using edition and analysis tools for the behavioral annex. J. P. Bodeveix et al. [11] propose an AADL behavioral annex and a technique to perform compositional real-time verification of AADL models through the use of a method which translates environmental constraints into behavior.

Other papers, such as the latter one, include the behavioral annex in their verification process of AADL models. B. Berthomieu et al. describe in [12] a formal verification tool chain for AADL with its behavioral annex available in the Topcased environment. They translate the AADL model to Fiacre and verify the behavior with a Time Petri Net Analyzer (Tina).

C. Ponsard et al. explore in [13] the interplay of requirements and architecture in a model-based perspective by defining a mapping and a constructive process taking into account specificities of embedded systems, especially the importance of non functional requirements. To generate the behavioral part of a system they first generate a finite state machine and then an AADL mode-transition.

A way to tackle AADL and its behavioral annex is translation to another language. To allow simulation M. Yassin Chkouri et al. propose in [14] a translation from AADL models into BIP models. They take into account behavior specifications allowing state variables, initialization, states and transitions sections to be defined and translating them into BIP. DUALLY [15] is an automated framework that allows architectural language interoperability through automated model transformation techniques. I. Malavolta et al. analyze the feasibility of integrating AADL and OSATE in DUALLY. They map AADL behavioral annex sections of states, composite states and transitions.

After analyzing the state of the art, a behavioral annex to the AADL standard appears to be necessary, which could be included in an AADL model in order to express the behavior of the components.

However, no approach uses SystemC [16], which is the recognized standard for modeling HW/SW platforms, with its great potential for integration of processors, buses, memories and specific platform HW. Our solution makes HW/SW co-design easier because of the use of SystemC.

SCoPE [17-18] is a C++ library that extends the standard language SystemC without modifying it. It simulates C/C++ SW code based on two different operating system interfaces (POSIX [19-20] and MicroC/OS). Moreover, it co-simulates these pieces of code with HW described in SystemC.

In a previous work [21], a preliminary version of AADS supporting a part of the AADL standard was developed. Now we have improved AADS to take into account the most important issues of the AADL behavioral annex (states, transitions, sending and receiving messages, etc.). AADS+ supports AADL behavioral annex simulation in SystemC, thus enabling the HW platform to be modeled and permitting HW/SW codesign. The AADL model is based on POSIX, so it supports many different RTOSs.

3. Previous work

AADS is written in Java and was developed as a plug-in [22] of Eclipse [23]. AADS enables the modeling of a subset of AADL for purposes of implementation and simulation. The starting point of the simulator is a functional AADL specification without detailed code. For each component, the corresponding timing constraints are defined. This initial AADL specification supports the verification of the global performance constraints of the system based on the specific timing constraints of the different components. The AADL model is parsed using AADS and a model suitable for simulation with SCoPE is produced, in order to check whether the AADL constraints are fulfilled. As the design process advances and, on the one hand, the actual functionality is attached to the SW components using the corresponding source code and, on the other, the functionality is mapped onto specific platform resources, a more accurate performance estimation is performed. These refined properties can be added to the AADL model and a new model can be generated by
AADS. By comparing the initial timing constraints with these refined, timing estimations, it is possible to verify the non-functional correctness of the design process at any refinement step.

AADL enables the specification of both the architecture and functionality of an embedded real-time system. AADS translates both to SystemC (see Figure 1). It parses the AADL model so the functionality is translated to an equivalent POSIX model and the architecture is represented in XML [24].

4. Translation of the behavioral annex

The AADL behavioral annex improves the specification of a component’s behavior. AADS+ parses the AADL model so the annex behavior specification sections are translated to an equivalent POSIX model.

The behavioral annex describes a transition system (an extended automaton) using optional sections:

- **State variables.** The state variables section declares typed identifiers. Types are data classifiers of the AADL model. AADS+ translates these state variables declaring variables with their corresponding type in the C++ source code of the thread or subprogram itself.

- **Initialization.** The state variables must be initialized in the initialization section using a sequence of assignments. AADS+ translates this initialization initializing the variables with their corresponding value where they were declared.

- **States.** The states section declares automaton states which can be qualified as initial, complete, return, urgent or composite. AADS+ uses this section to know which states have been defined.

- **Transitions.** The transitions section defines system transitions from a source state to a destination state. The transition can be guarded with events or Boolean conditions. An action part can be attached to the transition. It can perform subprogram calls, message sending or assignments. AADS+ translates the transitions section into switch and case statements to transit from one state to another. It starts in the initial state and moves to the next state when the guard of the transition is true. So the guard of the transition translated by AADS+ acts as a condition to execute the sentence/s of the state and to change the state. This sentence/s is the action of the transition translated by AADS+. If there is no guard there is no condition to check. The guard can be an expression as simple as on i < 5, so AADS+ will translate it directly.

Depending on the content of the guard and the action of the transition, AADS+ translates them into the corresponding sentences of source code.

- **Sending / receiving messages.** Messages are sent / received through event or event data ports. If p is an input port, p? de-queues an event port variable, p?x de-queues a datum on an event data port in the variable x. If p is an output port, p! calls Raise_Event on an event port, p!d writes data d in the event data port and calls Raise_Event.

In the first case the guard of a transition is p1?x (where p1 is an in event data port) and the action of that transition is p2!(x+1) (where p2 is an out event data port). AADS+ translates this case, checking whether a variable arrives at the POSIX message queue associated with the port p1. Then the variable is sent through the POSIX message queue associated with the port p2, in this case after adding 1 to it.

In the second case the guard of a transition is p1? (p1 is an in event port) and the action of that transition is p2! (p2 is an out event port). AADS+ translates this case, checking whether the corresponding POSIX signal associated with the port p1 has been received. Then the corresponding POSIX signal associated with the port p2 is sent.

- **Subprograms.** A behavior expressed by the annex can be attached to a subprogram implementation. The behavior can refer to the subprogram parameters and to variables. The automation specifying the subprogram implementation has one or more return states indicating the return to the caller. While the AADL control flows define the call sequences produced by a subprogram, the annex enables the expression of dependencies between the control flows and state variables or parameters. A subprogram specification can express other calls or notification of events.

In the first case the guard of a transition is p1? (p1 is an in event port) and the action of that transition is subp! (subp is a subprogram). AADS+ translates this case checking whether the corresponding POSIX signal associated with the port p1 has been received. If the signal has been received then the corresponding previously defined subprogram is called.

Parameters can be passed to called subprograms. The action of that transition could be subp!(5->x,2->y) where x and y are two in parameters of the subprogram.
subp. Then AADS+ translates it into a call to the 
subprogram with those two parameters as subp(5,2).

Using the AADL behavioral annex, it is possible to 
indicate in the action of a transition that the out 
parameter of a subprogram is the in parameter modified 
in some way. It could be po!(pi+1), where po is the out 
parameter and pi the in parameter. AADS+ translates 
this case, creating the source code in the subprogram 
that sums one to the in parameter and assigns the result 
to the out parameter.

In the last case the guard of a transition is on pi (pi 
is an in parameter of a subprogram) and the action of 
that transition is a call to a standard function like 
std::cout. To translate this transition AADS+ generates the C++ source code that checks whether the 
in parameter is true and, if it is, calls the standard 
function cout.

**Control structures.** Control structures support conditional execution of alternative actions (if, else, end if), conditional repetition of actions (while), and 
application of actions over all elements of a data 
component array, port queue content, or integer range 
(for). The For structure represents an ordered iteration 
over all elements. Within the for structure the element 
can be referenced by element_variable_identifier, 
which acts as a local variable with the name scope of 
for structure.

In the case that the action of a transition contains a 
conditional structure of the type: if (logical value 
expression) behavior_actions [else behavior_actions] 
end if, AADS+ translates it producing the source code 
with the analogous if else structure in C++, adapting 
the differences between them.

The same can be said about for and while structures 
of the type: for (element_variable_identifier in values) 
{behavior_actions} and while (logical value 
expression) {behavior_actions}. AADS+ translates 
them producing the source code with the analogous for 
and while structure in C++, adapting the differences 
between them.

**Arrays.** To declare collections of data which are 
considered to be ordered the notion of multiplicity is 
used. AADS+ translates multiplicity into a C++ array 
of data. The type of the array is the same in both 
AADL and C++.

## 5. Case study

The proposed method implemented in AADS+ has 
been tested in a typical case study, the cruise control 
presented in Figure 2, to assure the feasibility of the 
translation. Cruise control is a system that 
automatically controls the velocity of a motor vehicle. 
The driver sets a speed and the system will take over 
the throttle to maintain it.

The use of the AADL behavioral annex with 
AADS+ has been validated through the refinement of 
the original cruise control design. As the original 
model was developed without using the behavioral
annex, the model lacked relevant behavioral information. The annex overcomes these problems and enables the development of a more detailed architecture.

The figure shows an AADL model with its behavioral annex of a cruise control system, taken from the collection of AADL examples in the OSATE release, but modified to add some subcomponents. The system component contains two processors, two memories and two devices connected by a bus, and two SW subsystems. Each of the subsystems is bound to a separate processor and to a separate memory. Threads communicate via data ports, event ports and event data ports. Some data access connections can be seen too. There are some subprograms within threads and within data subcomponents and the call sequences (local and remote) between them are shown. The parameter connections between subprograms are shown too. One subsystem has two processes, one with four threads and the other with one. The other subsystem contains one process, with two threads.

The files produced by AADS+ are compiled with SCoPE to simulate the model. The results obtained in the simulation are used to refine the model of the cruise control as needed.

Figures 3 and 4 are an example of the translation performed by AADS+ of the behavior specification of a thread. Messages are sent and received through event data ports. In this case the guard of a transition is \( \text{Refspd} \text{Mph} ? x \) and the action of that transition is \( \text{Filrefspd} \text{Mph}(x+1) \). AADS+ translates it checking whether a variable arrives at the POSIX message queue associated with the port \( \text{Refspd} \text{Mph} \). Then the variable is sent through the POSIX message queue associated with the port \( \text{Filrefspd} \text{Mph} \), after adding 1 to it.

Subprograms with their behavior specifications have been added to the AADL model of the cruise control to obtain the desired system performance. For example, to detect if a button has been pushed by the driver the corresponding behavior was added to a subprogram in \( \text{Button} \text{panel} \) thread and refined through simulation.

When the driver activates the cruise control, an event is sent to the \( \text{Refspd} \text{thread} \) that sends another event to the \( \text{Instrument} \text{panel} \text{thread} \) to show the activation; this behavior has been implemented in the thread \( \text{Refspd} \).

```c
thread Cruise1
features
...
...;
end Cruise1;

thread implementation Cruise1.Simulink
properties
...
...;
end Cruise1.Simulink;

Figure 3. A behavior specification of a thread.
```

```c
float mens;
int len;
unsigned int pri;
len = mq_receive(\text{Refspd} \text{queue}, (char *)&mens, sizeof(mens), &pri);
if (len > 0) {
    cout << "Cruise1 receives message from " +
         "\text{Refspd} containing " << mens << endl;
    \text{XRefspd} \text{Mph} = mens;
    \text{XRefspd} \text{Mph} = 1;
...;
    int stateCruise1 = 0;
...;
    switch (stateCruise1) {
        case 0:
            if (\text{XRefspd} \text{Mph} ) {
                float mens = XRefspd \text{Mph}+1;
                cout << "Cruise1 sends a message " +
                     "containing " << mens << endl;
                mq_send(\text{Filrefspd} \text{queue},
                         (const char *)&mens, sizeof(mens),1);
                \text{CRefspd} \text{Mph} = 0;
                stateCruise1 = 1;
            }
            break;
        case 1:
            break;
        default:
            break;
    }
```

Figure 4. C++ code translated by AADS+.

The correct operation of the behavior specification created to know whether the \( \text{Driver} \text{model} \text{logic} \) is activated or disactivated was refined by simulating the model.

Another example of behavior specification is in the \( \text{thread} \text{ Instrument} \text{panel} \), which provides information
about the actuation zone depending on the speed of the vehicle (see Figure 5). All these examples were translated by AADS+ in an analogous way as can be seen in the previous figures.

![Figure 5. State diagram of one Instrument panel's behavior.](image)

Refinement of the original cruise control model with behavior specifications does not require a large number of AADL code lines, AADS+ does not produce so many C++ code lines as one might fear (see Figure 6) and the gain in expressiveness of the model’s behavior is great. Furthermore, the cost in terms of use of CPU, core energy/power, bus access time, etc is slight.

![Figure 6. Comparison between the two models' metrics.](image)

### 6. Conclusions and further work

This paper presents the simulation of the AADL behavioral annex using the AADS+ simulation tool. AADS+ supports the refinement of AADL models, including the behavioral annex, through performance analysis done with SCoPE, after translating those models.

The generation of the SystemC model from the AADL specification and its behavioral annex is not straightforward. Nevertheless, the SystemC model generated by AADS+ is able to capture the fundamental dynamic properties of the initial system specification. In this way, AADS+ supports design space exploration by refinement of the AADL functionality and its implementation on an optimized platform.

Future work includes incorporation of AADS+ features that appear in V2.0 of the AADL standard. Furthermore, the source code produced by AADS+ for the software components will be made compatible with the ASSERT Ravenscar Computational Model (RCM).

### 7. References


