Simulation of system architectures with AADL

J.-F. Tilman[1], R. Sezestre[2], A. Schyn[3]

1: Geensyde, 242 bvd Jean Jaurès, 92100 Boulogne-Billancourt, France
2: Geensys, 242 bvd Jean Jaurès, 92100 Boulogne-Billancourt, France
3: Axlog, 19-21 rue du 8 mai 1945, 94110 Arcueil, France

Abstract: AADL is a language dedicated to the modeling of system architectures. Among all the possible analyses, verifications and other usages of such models, this article considers the simulation of the behavior of these systems. After an overview of the context and the illustration of the interest of such a simulation, it presents the development of ADeS, a simulator which aims at representing the whole behavior information provided by the AADL standard. The different behavioral aspects to be considered are presented, followed by the main technical choices of the development of ADeS and the difficulties which appeared. The article ends with consideration on the current status and the perspectives of the tool for the AADL community.

Keywords: AADL, simulation, embedded systems

1. Introduction

The current system engineering practices are more and more based on modeling approaches and the use of architecture description languages (ADL). This evolution is driven by the need for stronger methods to handle the increasing complexity of the embedded systems. The use of ADLs is coupled with techniques and tools to help in the development: automatic generation, performing verifications, etc.
One of the main ADLs currently considered by industry in system engineering for embedded systems is AADL [1]. This language provides a means to model both the software and the execution platform architectures. In parallel with its standardization process, efforts have been spent to develop tooling support, based on various approaches. We will consider in this paper the simulation of the behavior of a system described with AADL.
First, let us consider the interest of the simulation of AADL models, the context of this question, and some cases which illustrate its use in existing tools. Second, we will deeper consider how such a simulation is performed in ADeS, a tool which aims at supporting the whole behavioral aspects of AADL. Last, we will deal with the integration of such a tool in the AADL environment and the perspectives it offers for the future.

2. Context of the study

2.1 AADL overview

AADL is now better known by the embedded system community, and the purpose of this paper is not to deeply present it once again. However, few words to introduce its main concepts are useful to understand the specificity of a simulation based on this language.

An AADL model is composed of a set of *components*. Each component belongs to a *category* (e.g., processor, thread, subprogram, system...) which has a precise semantics. It is described in two parts: the *type* represents its interface with outside, and the *implementation* represents its contents. These components are hierarchically composed and connected together to form a complex architecture. Such a description may be enriched by associating valued properties to detail many aspects of the description. For instance, a thread will receive a deadline property or a compute execution time property.

AADL provides operational *modes* to represent various configurations of a same system, and transitions between these configurations. Depending on the current mode, components may be activated or deactivated, connections may change, properties may have different values, etc.

Last, AADL supports an *annex* mechanism to extend the description capabilities of the language by introducing a dedicated sub-language. A *behavior annex* is currently defined by the standardization committee.

2.2 Verifying properties of a model

Once we have a means to produce a precise model of a system, we may want to verify properties of this model. Some of these properties can be checked by applying formal techniques. For instance, the schedulability of a set of tasks can be ensured, thanks to formulas provided by the scientist literature. Provided that information is available in a model, it is possible to implement these formulas in a tool [2]. Other verification techniques are based on model checking. Several teams use these approaches to verify various kinds of properties, and develop tools implementing their solutions. An example is Cheddar, which is designed for checking task temporal constraints and buffer sizes of real time applications and systems [3].

Ensuring the schedulability of a reduced set of periodic tasks by formal techniques may be reasonable. When the size and the complexity of the model grow, with many interactions between tasks, or dynamic changes of the configuration, the problem becomes more difficult, sometimes impossible. In such a case, in absence of proofs, we may want to see what happens in the system by simulating its behavior. This is illustrated by Cheddar, which uses scheduling and buffer simulation when its feasibility tests can not be applied. BIP, developed by Verimag, or the Furness Toolset, proposed by Fremont Associates, also encompass simulation capabilities, even if they only consider subsets of AADL [4, 5].

2.3 Simulation from the AADL model

More generally, the designer of a system is interested by animating his model. This provides him an overview on how his system will behave when finished. This helps him, for instance, to better dimension the system, or to detect locks, missed deadlines and other problems – even if a simulation will never replace other verification techniques to ensure the absence of failures.

Another approach to use simulation consists in computing the evolution of the system in batch mode and recording the results, and then analyzing these results during a second step.

However, AADL is *a priori* designed to describe static system architectures. Thus it may seem strange to envision the simulation of such a model. In fact, compared with other modeling languages, and thanks to its precise semantics which includes the description of behavioral aspects, AADL makes possible such simulations, as we will see later.

2.4 Lack for a full support of the AADL behavior

Most of the AADL tools which use simulation techniques do it to help in the verification of particular properties. They generally not consider the whole AADL language. The restrictions may be due to an incomplete support of all the possible constructs of the language. In this case the tools only understand a subset of the language, or just ignore some possible constructs.

Sometimes, the tools do not respect the exact standard behavior associated with the elements of the architecture, as specified by the AADL standard. Either they use a simpler model, or they use their own behavior model.

In this context, there was no tool, at our knowledge, which was able to simulate the whole behavior of an AADL description with respect to what is specified by the standard. This is one of the main reasons of the development of ADeS, an AADL tool which aims at simulating the full behavior of an AADL system architecture.

2.5 Elaboration of the AADL standard

The development of the first prototype of ADeS started at Axlog Ingénierie, where the authors were all employed, in the context of a research project with the European Space Agency (ESA) to qualify the interest of the future AADL language for space domain.

The motivation of this development was the assertion of the feasibility of tools supporting AADL, but also the need for feedbacks to the standardization committee where we were involved in the definition of the language. At this time, it was about the only AADL tool, and its development allowed the detection of many problems in the draft grammars of the language.

The creation of OSATE, the *open source AADL tool environment*, by SEI has been the opportunity for a completely new version of the simulator [6]. Since OSATE consists in a set of Eclipse plug-ins which may be reused to take advantage of their functions, ADeS became also an Eclipse plug-in, and gave up its own AADL parser to reuse the services provided by OSATE.

More generally, the designer of a system is

3. Simulation of AADL in details

3.1 What has to be simulated

When considering the AADL description of a system, behavioral information comes from many locations. First, the standard of the language provides a precise description of each component category. For each of them, it explains the exact behavior. Some component categories are active, that is they represent an element of the architecture which executes something. The threads are the best example of such an active component. The specification of a thread behavior is complex by nature. Its complexity is reinforced by the fact that the AADL standards aims at not restraining too much the user, and make possible the description of all the common kinds of tasks which exist in the real time community. Thus, an AADL thread may be parameterized to match with the exact thread of the user. The global behavior of a thread is then the sum of many details which exist in actual real time threads. For instance it is possible to describe an initialization phase, an activation phase, or a recovery phase, even if this will not be used in some cases by the user.

The passive AADL component categories can also introduce behavioral information which has to be taken into account by the simulation. For instance, a data component, which may represent a global variable in memory, can also be used to represent a shared data. In such a case, specific mechanisms are used to manage this sharing. When simulating the access to this variable by a thread, the simulation has to add the representation of these mechanisms. Another example: a bus, which is used to support communications, introduces some constraints on the transfer of data or events.

When modes are used, they represent various runtime configurations of the system. For each of these configurations a specific behavior appears, as if we had several different systems. What has an impact on the simulation is the fact that these modes can change during the live of the system, and these changes appears during the simulation. When such a mode change happens, the simulation has to take into account all the consequences: a subcomponent may be removed, another one will appear, some connections are redefined, the values associated with properties change, threads are halted or restarted, etc. Some rules exist in the AADL standard to precisely specify when and how these mode changes happen.

The pure AADL specification may be completed by the use of standard annexes. The *error model annex* gives details on the handling of errors, the arrival laws of errors, etc. This should be taken into account, since it has an influence on the result of the simulation. The *behavior annex* is, of course, the major complement to refine the behavior description of a system. With this annex it is possible to explain how a subprogram, called by a thread, will work, raise events, etc. Supporting this annex in the simulation brings many capabilities to analyze the behavior of the modeled system.

The last source of information about the behavior is what the user may introduce by himself. Indeed, he can create his own new properties or annexes to represent what he wants. However, if he does so, he also has to develop the support to understand the semantics of his extensions. A simulator may provide extension points to make possible the development of plug-ins to support such user defined complements.

3.2 Simulation management

Several solutions exist to manage the execution of a simulation, and the choice of the appropriate simulation engine depends on the purpose of the simulation. An *integration* engine is adapted to simulations which contain only continuous variables. A *step-by-step* engine is adapted to simulations with no continuous variables, and where the events are implicit. In this case the engine tests at each periodic step the status of the events to detect when they are raised. An *event-driven* engine is adapted to simulations with no continuous variables, and where the events are explicit. Here, it is possible to directly jump from the execution of one event to the execution of the following, which is more efficient. Combined engines may also exist, to combine continuous and discrete variables, as well described in [7].

The simulation of an AADL architecture is clearly largely discrete: the elements have states, their properties change instantaneously, etc. All the events are explicit. Then, the simulation engine technique used for ADeS is naturally the event-driven approach. An AADL system will usually contain periodic tasks which will be regularly dispatched, but also aperiodic events which may happen at any time. Thus, the amount of things to be simulated may vary in the time, with long empty durations and other heavy periods. Thanks to the event-driven approach, the performances of the simulation are improved: when nothing happens, the simulated time is immediately advanced up to the next event.

Concretely, the kernel of the tool works as a scheduler for the simulation events. Each event has a date when it has to be executed, and is added into an ordered list. When two events have the same date, they can have different priority levels to order them. The execution of an event generally produces new events which will be executed later. For instance, when the task scheduler decides a switch between tasks, some events are raised to preempt the first task and start the second.

However, we identify two drawbacks of this solution. First, the simulated time may advance at an irregular speed, depending on the activity of the system. If we want to present a "real time" simulation to the user, we have to synchronize this simulated time with the actual time. This is easily feasible by adding specific periodic synchronization events. The second drawback might appear if we imagine, in the future, an extension of the simulator to take into account the environment in which the studied system evolves. In such a case, continuous variables may be useful. Solutions exist, either by extending the simulation engine, or by accepting compromises in the representation of these continuous variables in the simulation.

3.3 Layered structure

Simulating the complex behavior of a full system involves several aspects. Some of them are generic, other ones dependent on the AADL specificities. To cope with this reality, the simulation tool has also to be organized into layers.

A first set of layers, called “jimex” is in charge of the pure simulation aspects. It is organized as follow:

- The *jimex core* component implements the lowest level of the simulator, the simulation engine. It defines simulation events and manages them. It is completely independent from any specific purpose of the final simulation. At this level the simulation events are just characterized by the date when they have to be raised and their priority;

- The *jimex base* component provides higher level simulation elements, and particularly more specialized events, as required by the simulation of elements of an embedded system;
- The *jimex aadl* component introduces all the AADL specific semantics, as defined by the AADL standard. However, it is still independent from OSATE, and might be used to support an AADL simulation related to another modeling tool.

The other layers play a role in the management of the simulation:

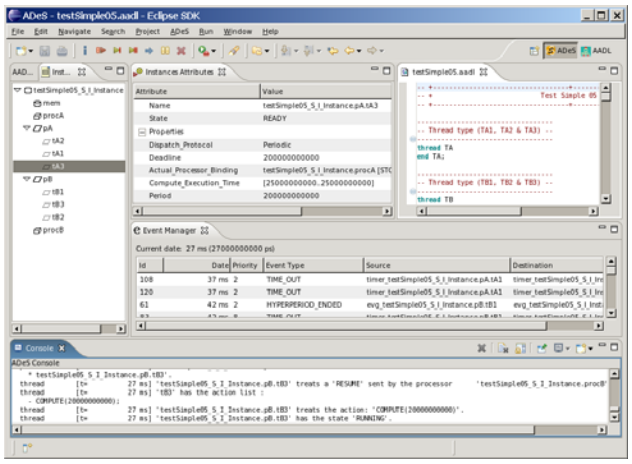
- The *ades instantiation* component implements the mechanisms to build the simulation elements from the AADL elements as described in OSATE;
- The *ades trace* component is in charge of recording all what happens during the simulation and providing this information to the upper levels in charge of the man-machine interface;
- The *ades simulator* component implements the man-machine interface which control the simulation and displays the results.

3.4 Exploitation of the results

Several ways exist to exploit the results of a simulation: observation during the computation, compilations and analyses of the resulting data, production of reports, etc. Most of them are based on the systematic record of all what is computed by the simulator, in order to keep a trace afterward. Such a trace mechanism is integrated into ADeS. Each element of the AADL architecture records what happens for it: emission of an event, change of a property value, transmission of a data onto a bus... All this information is stored into an XML file. The choice of this format has been done to make easier post-analyses by anybody.

When coupling the trace coming from a run of the simulation with information on the structure of the simulated architecture itself, it becomes possible to rebuild the exact state of a simulation at a given date in the past. Such a *snapshot* of a simulation makes possible the replay of a simulation, either to refine the analysis of an interesting point, or to change parameters and compare two different runs without playing twice what is similar before the divergence. ADeS is able to store these snapshots and reload them to propose this capability. This mechanism is also used at the beginning of the simulation. Indeed, the creation of a simulation, before its first run, consists in the creation of such a snapshot at *t*=0 and its loading.

During the execution of a simulation, the user may observe the status of all the properties of the elements composing the architecture. However, this information is presented as a table. Effort is still needed to improve the graphical rendering of the results, for instance by proposing chronograms and other high-level widgets to give a complete and intuitive view of the results. Figure 1 shows the main window of the tool, with several views to control the current values of the parameters, the events of the simulation, the trace of the results.



Attribute	Value
Name	testSimple05_S_I_Instance.pA.tA3
State	READY
Properties	
Dispatch_Protocol	Periodic
Deadline	200000000000
Actual_Processor_Binding	testSimple05_S_I_Instance.procA [STC
Compute_Execution_Time	[25000000000..25000000000]
Period	200000000000

```
-- +-------------------------------------+
-- +                        Test Simple 05
-- +-------------------------------------+

---------------------------------
-- Thread type (TA1, TA2 & TA3) --
---------------------------------
thread TA
end TA;

---------------------------------
-- Thread type (TB1, TB2 & TB3) --
---------------------------------
thread TB
```

Current date: 27 ms (27000000000 ps)

Id	Date	Priority	Event Type	Source	Destination
108	37 ms	2	TIME_OUT	timer_testSimple05_S_I_Instance.pA.tA1	timer_testSimple05_S_I_Ins
120	37 ms	2	TIME_OUT	timer_testSimple05_S_I_Instance.pA.tA1	timer_testSimple05_S_I_Ins
61	42 ms	2	HYPERPERIOD_ENDED	evg_testSimple05_S_I_Instance.pB.tB1	evg_testSimple05_S_I_Inst

```
  * testSimple05_S_I_Instance.pB.tB3'.
thread       [t=          27 ms] 'testSimple05_S_I_Instance.pB.tB3' treats a 'RESUME' sent by the processor      'testSimple05_S_I_Instance.procB'
thread       [t=          27 ms] 'tB3' has the action list :
  - COMPUTE(20000000000);
thread       [t=          27 ms] 'testSimple05_S_I_Instance.pB.tB3' treats the action: 'COMPUTE(20000000000)'.
thread       [t=          27 ms] 'testSimple05_S_I_Instance.pB.tB3' has the state 'RUNNING'.
```

Figure 1: Snapshot of the ADeS main window

3.5 Complexity and openness

When considering all the aspects composing the behavior of an AADL architecture, we have to deal with a high complexity. This complexity comes not only from the fact that many details have their own rules that have to be taken into account, but also from the impact they may have onto the global behavior. As usual, passing from a local level to the global level introduces complexity.

The best illustration of this difficulty is the impact of the modes onto the behavior. As explained before, many elements of the architecture may have modes, which control some parameters, enable or disable subcomponents, change connections, etc. Since such an element having modes may be integrated as another subcomponent with its own modes, and so on up to the top of the hierarchy, we have to consider the *system operation modes* (SOM), which are the combination of all the modes of all the components. The set of possible SOMs is then the cross product of the sets of modes for each component. We have here a possible combinatorial explosion.

The complexity of the simulated behavior must not be an obstacle to the openness of the simulator. Since AADL supports the introduction of user-defined extensions, represented by annexes, an

AADL simulator has also to support such possible extensions. Fortunately, the modular approach adopted to build ADeS helps in this support. The semantics of an extension defined by a user will only be known by this user; thus, ADeS may offer extension points and mechanisms to connect plug-ins, but in any cases, the development of these plug-ins is of the responsibility of the user.

4. ADeS and the AADL environment

4.1 Integration with other AADL tools

As mentioned before, OSATE is an AADL tool which provides a textual modeler, a parser, and all what is needed by any tool to handle the data structure representing the AADL architecture. It is designed as a set of Eclipse plug-ins, and many other tools have been developed on top of it to bring their own features: analyses, verifications, etc. OSATE is now integrated into the Topcased environment, also based on Eclipse, and which provides graphical modelers. We assist at the emergence of a complete tool environment around AADL. In this context, the developer of a new AADL tool has to wonder whether it is expedient for him to integrate his tool into this environment.

This integration has been done for ADeS. The first benefit is the reuse of the AADL parser and the underlying data model, which is shared with other tools. The second benefit is the capability for the user to have all its tools at the same place. He uses textual and/or graphical modelers, verifies the completeness and consistency of his model, perform various analyses, statistics and validations, and run his simulation without any translation of his models.

4.2 Project management

As a consequence of this integration, the management of simulation projects has to be done in the sense of Eclipse projects, and also in a compatible way with the projects of the other AADL tools.

In ADeS, a simulation project is created in relation with a preexisting *AADL project*, provided by OSATE and used to model the target architecture. The simulation project may contain one or more simulations of the same system, characterized by different scenarios. Once a simulation is created, it may be executed, and at any moment a capture of its state may be done to be replayed later. This capture, also called *snapshot*, constitutes a new simulation in the project.

A synchronization mechanism is able to detect changes in the AADL project and take them into account in the simulation project. Generally, modifications of the model make the existing simulations obsolete.

4.3 License and availability

Another consequence of the integration of ADeS into the OSATE/Topcased tool family is the need for a free open-source license. The chosen license is the Eclipse public license (EPL).

Now, the ADeS source repository is hosted by Topcased and is available for downloads and contributions[1].

4.4 Results and perspectives

ADeS has not yet been extensively used on large models. Feedbacks are still missing to get a correct overview of its results. However, experimentations on smaller models show interesting capabilities, even if the analysis of the results may be a bit painful due to the lack of high level graphical displays. *Spices*, a European R&D project dedicated to predictable system engineering based on AADL, provides us the opportunity to continue the development of ADeS and trial it further.

The official behavior annex is currently standardized by the AADL committee. Since it was not stable enough during the development of ADeS, another minimal behavior annex has been implemented. This shown the capability to plug extensions to support user-specific annexes. However, we may expect to shortly support the future standardized behavior annex. Another standard annex exists, the error model annex. Its support is also missing for the moment.

The Topcased project, which now encompasses OSATE, also deals with other modeling languages. It needs a generic simulation support, able to cover multi model formalisms. In the future, ADeS could be adapted to such a larger purpose, or its technology may be reused in with this new objective.

Another important and positive result of ADeS is for the AADL standardization committee. Indeed, the development of this tool has been – and is still – the opportunity to experiment evolutions of AADL in parallel with the works of the committee and bring feedbacks on the lacks, inconsistencies and other problems in the draft versions of the standard.

5. Conclusion

This article has shown the interest for a simulation based on AADL models, and how this architecture description language provides a complex behavioral information, thanks to its semantics and the mechanisms it proposes. This approach is already used by several tools for verification purposes. ADeS has been designed to support all the details of this behavior, to provide an animation of the models as precise as possible. Its development, still in

1 http://gforge.enseeiht.fr/projects/ades/

progress, has accompanied the definition of the AADL standard and provided a strong feedback for this standardization. The integration of ADeS into the AADL tool suite and its open-source license may open new perspectives for its usage and development.

6. Acknowledgment

The works presented in this paper have been partially conducted during projects with ESA and other European project. The authors acknowledge the cooperation of some partners of these projects, and of the members of the AADL standardization committee.

7. References

[1] SAE, "*Architecture analysis & design language (AADL)*", November 2004.

[2] Christophe Guettier, Jean-François Hermant: “*Static Mapping of Hard Real-Time Applications onto Multi-Processor Architectures using Constraint Logic Programming*”, ICAPS, 2005.

[3] F. Singhoff, J. Legrand, L. Nana, and L. Marcé: “*Scheduling and Memory requirement analysis with AADL*“, ACM Ada Letters journal, 25(4):1-10, ACM Press. Also published in the proceedings of the ACM SIGAda International Conference, Atlanta, 14-17 November, 2005.

[4] Verimag: “*BIP*”, http://www-verimag.imag.fr/~async/BIP/bip.html

[5] Fremont Associates: “*Furness Toolset*”, http://www.furnesstoolset.com/

[6] SEI: “*OSATE*”, http://www.aadl.info/tool/osate.html

[7] C. Sautereau, J-P Rosen, "*ESCADRE V5 / Simulation, Manuel du concepteur*", 2002-08-13.

8. Glossary

AADL:	Architecture analysis & design language
ADL:	Architecture description language
EPL:	Eclipse public license
ESA:	European Space Agency
SEI:	Software Engineering Institute (Carnegie Mellon)
SOM:	System operation mode
XML:	extensible markup language