

The AADL behaviour annex – experiments and roadmap *

Ricardo Bedin França Jean-Paul Bodeveix Mamoun Filali Jean-François Rolland †
IRIT - Université Paul Sabatier
118 route de Narbonne
F-31062 Toulouse
{bodeveix,filali}@irit.fr

David Chemouil
Centre national d'études spatiales, Toulouse

Dave Thomas
EADS Astrium Satellites

Abstract

In this paper, we present an evaluation of the AADL Behavioural Annex that is currently in evaluation phase. We relate our experiment with respect to a development concerning the reengineering of a flight software. This experiment has led us to introduce hierarchical aspects and study the link especially with AADL modes. We discuss about the definition of a semantics for the AADL execution model and propose some enhancements.

1. Introduction

Architecture description languages take a growing importance for component based development of systems. They offer a support for a design methodology that promotes the hierarchical decomposition of a system into interacting entities which may be software (modules, processes, threads, ...) or hardware (processor, bus, ...) components. This high level specification provides a framework to perform early verification of the system and ensure composability of separately developed parts. This is even more important when real-time systems are considered. In this context, we present the AADL language[2] and our proposal for a behavioral extension which allows a more detailed specification of the software behavior. This extension, incorporated to AADL via its annex mechanism, is validated through the reengineering of a spacecraft flight software previously developed in AADL [9] using the Topcased/OSATE environment[10, 6]. This study helps us in improving the expressive power of the annex and leads us to propose evolutions of the annex and the AADL execution model.

*Study partly founded by CNES

†Work founded by CNES and EADS Astrium Satellites

2. Overview of AADL and its behavioral annex

AADL (Architecture Analysis and Design Language) is a standard developed by the SAE based on the MetaH language[11]. Its main purpose is to describe the dynamic architecture of an embedded system, its real-time constraints and the mapping of software to hardware components. It is then possible to perform real time checks [8] at an early stage of the development of a system. An AADL specification defines various kinds of software and hardware component types (systems, processes, threads, processors, buses, ...), their real time properties (period, WCET, ...) and how they interact. Software components export features such as event, data and event data ports, shared data accesses, server subprograms. Component implementations are either defined using any source language (Ada, C, ...) or using instances of existing component types and connections between their features, as with any ADL. The architecture of a component implementation may be mode dependent. A mode automaton specifying mode changes on event occurrences is then added.

A set of properties can be attached to each AADL declaration (real-time information, communication protocols, ...). Model analysis tools exploit such properties to perform schedulability, flow latency, memory consumption analyses, ... and to propose thread bindings to processors which makes the system schedulable and such that communications are minimized[6].

AADL is an extensible language: external languages can be used to define annexes. This is how we have defined a behavioral annex to specify the detailed behavior of threads in order to perform model checking and advanced code generation. A second extension mechanism is via the definition of new properties. AADL analysis tools are based on annotations of the AADL models through properties. Thus, a tool can provide a set of property definitions that the user

will exploit to enrich its model.

2.1. AADL execution model

AADL execution model mixes synchronous and asynchronous aspects. A synchronous execution model is obtained by considering logically synchronized periodic threads communicating through data ports. Values are transmitted from output ports to input ports at the start of the period. An immediate transfer protocol exists and allows communication between two threads within the same period, which implements the zero-time computation hypothesis. Real-time properties attached to model elements provide a means to check the validity of the synchronous hypothesis. Asynchrony is introduced when using the full power of AADL: it is possible to declare buffered data, to raise events on event or event data ports, to specify sporadic and aperiodic threads with different periods, communication through shared variables and remote procedure calls.

The execution model is (partially) formalized in the standard. It is based on a restricted use of hybrid automata modeling stopwatches. They define the execution time of threads and compare it with their required deadlines. However, the extensive formal specification of AADL has not been done yet.

2.2. AADL Behavioral Annex

AADL does not support by itself the expression of detailed behaviors. At the best, it is possible to specify the non-deterministic behavior of a thread as a set of sequences of subprogram calls. The behavioral annex can be seen as a refinement of these call sequences where the choice between call sequences will be made explicit. The overall structure of the behavioral annex is a non-deterministic hierarchical automaton using an AADL-like mode automaton syntax: 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 it is attached. Its semantics closely depends on the AADL execution model:

- When dispatched, the execution of the component starts on an *initial* state.
- A thread completes its execution after reaching a *complete* state and executing the action part of the transition. Subsequent dispatches will start from that state.
- A state can be declared *composite*. A subautomaton is attached to such states. A transition can exit a substate and have as target a descendant of any ancestor state of the composite state containing it.

- The annex has access to the contents of input ports as there where at the time of dispatch. Subsequent received data are not accessible.
- Data written on output data port is transmitted after completion.
- Events and event data are explicitly sent by specific actions.
- Asynchrony is supported through events, access to shared data via AADL data access declarations and remote procedure calls.
- Real-time aspects are supported by three primitives. `delay(min,max)` specifies suspension during a non-deterministic amount of time. `computation(min,max)` abstracts a computation by its non-deterministic CPU consumption. A timeout construct can be used within guards.

Declarations (subcomponent instances, connections, property values, ...) introduced in the implementation of an AADL component can depend on the current mode of the component. It is thus necessary to provide a means to specify mode dependant behaviors. This has been made possible by considering AADL modes as states within the annex. Consequently, the annex extends AADL mode automata by associating guards and actions to mode transitions. Furthermore, if the mode state is declared to be composite, a mode relative automaton can be defined.

As in statecharts, annex states as well as AADL modes can be declared concurrent and refined by a set of regions. Each region contains a possibly hierarchical automaton. The current state of the concurrent automaton is defined to be the set of current states of each sub-automata. Sub-automata become active if a transition targets the concurrent state or transitions from a *fork* state target states within each region. The concurrent machines terminate if a transition starting from the concurrent state can be fired, or if all transitions starting from states of each region and targeting a *join* state can be fired.

Region automatons are composed in parallel but, to conform with AADL specification, region automatons are not associated with threads. They allow a more compact specification of a non-deterministic sequential behavior.

3. The ArchiDyn project

The ArchiDyn project [9] consisted in the study of methods to model the dynamic architecture of spacecraft flight software. The project employed the AADL language to create specifications of a flight software, corresponding to certain steps of the application development cycle. Each

specification was analysed according to its purpose, in order to evaluate how well AADL can answer the designers' needs. In addition, a model-based design methodology has been proposed to aid in the development of flight control software. The project has taken as example the Pléiades satellite. It is a low earth orbit, high-resolution satellite, with all its construction based in its single observation instrument. The Pléiade HR satellite first launch is scheduled for 2008.

3.1. The On-Board Software

The On-Board Software (OSW) consists basically in a central processor connected to other equipment via 1553 (MIL-STD-1553B) buses. In order to ensure modularity, the whole software was decomposed in four main parts:

- System Application, that coordinates the other modules.
- Attitude and Orbit Control System (AOCS).
- Platform, which manages the temperature and electrical power in the satellite.
- Payload, which manages the mission aspects.

Each one of these parts has a periodic task which commands several aperiodic tasks, and all the periodic tasks operate at the same frequency.

The modeling has been made with the standard AADL syntax and the Behavioral Annex. Three different AADL specifications were designed, to represent different aspects of the system's architecture and accomplish definite objectives:

N0 Behavioral and functional high-level, platform-independent architecture, used in behavioral analysis.

N1 Logical dynamic architecture, used mainly in schedulability analysis.

N2 Concrete dynamic architecture, used in lower-level studies such as buffer dimensioning.

Due to confidentiality reasons, this paper does not present the N2 specification of the Pléiades OSW.

3.2. N0 – Functional architecture

At this level, the behavior of the software is described, as well as the input/output interfaces that represent the data flow through the set of high-level components. All the modules are represented with AADL `system` components, to ensure the needed flexibility at this level of abstraction. The mode automata of each module were represented with the

standard AADL syntax when possible, and also with the Behavior Annex.

A pure AADL modeling of the N0 level was uneasy due to some lack of expressiveness of the language:

- There is no adequate way to represent hierarchical mode automata. Hence, some subcomponents should be created to represent sub-modes, which is clearly unnatural and is a major drawback to the use of pure AADL in this level of abstraction.
- The standard AADL mode automaton does not have the possibility of triggering an action in a mode transition. To address this problem and represent some procedures that were meant to be triggered in a mode transition, the Behavioral Annex must be used. This annex also helps to solve another problem seen in the AADL mode automaton: the lack of "global" transitions – those enabled from multiple states to another state – which forced several similar transition declarations.

3.3. N1 – Logical Architecture

The N1 AADL specification describes the threads which constitute the system, along with their properties and synchronization events. Such synchronization are represented with the use of the Behavioral Annex, since the AADL standard has no adequate mechanisms to represent suspended tasks. This specification has been used together with the Cheddar [7] analysis tool for schedulability analysis, but some adaptations in the model are necessary to be correctly interpreted by the analysis tool – for example, some tool-specific properties must be defined.

4. The Development Process

4.1. Relations among modeling levels

The relation between N0 and N1 cannot be seen directly from looking at the models, because many lower-level aspects are implicit in functional specifications – for example, the modules whose concurrency is clearly seen with the N1 threads are represented with systems in N0, without any concern about representing explicitly their parallel execution. Hence, one should construct a table to create the relations that link functionalities from N0 to the N1 threads. The link from N1 to N2 is more easily seen, since N2 can be treated as a refinement of N1.

4.2. Modeling Patterns

The modeling of the Pléiades software also intended to create a framework to design other flight software. In the

architectural level, all the modules of the Pléiades OBSW were modeled following a pattern of one master periodic task commanding other aperiodic tasks. In the AADL specification there was also the use of some basic patterns, as in Figure 1.

```

package PATTERN public
system EQUIPMENT_CONTROL
  features
    ON_REQ: in event port;
    OFF_REQ: in event port;
  end EQUIPMENT_CONTROL;

system implementation EQUIPMENT_CONTROL.N01
  modes
    OFF : initial mode;
    ON: mode;
    OFF -ON_REQ-> ON;
    ON -OFF_REQ-> OFF;
  end EQUIPMENT_CONTROL.N01;
end PATTERN;

```

Figure 1. A reusable generic component

With such patterns, other components may “extend” the pattern component and aggregate their own characteristics. Support for architectural patterns will be greatly improved in futures versions of AADL.

5. Behavioral specification of the Pléiades On-Board Software

This section presents in more detail the N0 and N1 levels of the flight control software. They provide two use cases for the behavioral annex, which leads to feedback for future enhancements of AADL and the annex. The first level illustrates the specification of mode dependant behaviors while the second level exploits concurrent states and real-time aspects.

5.1. Functional specification (N0)

The functional specification proposed by the N0 level is essentially made of hierarchical modes, which are not supported by AADL. As previously mentioned, a pure AADL modelling was not adequate. The N0 level requires a hierarchical decomposition of modes, which is supported by the behavioral annex we have proposed for AADL. Figure 2 represents the modes automaton of the AOCS module.

The main modes, after the system is turned on, are the Acquisition and Safe Hold (ASH), Normal Mode (NM) and Orbit Control Mode (OCM). The ASH and NM modes have sub-modes, which complicate their representation in standard AADL syntax. The behavioral annex extends AADL mode automata and allows the association of actions to mode transitions. Furthermore, modes can be hierarchically

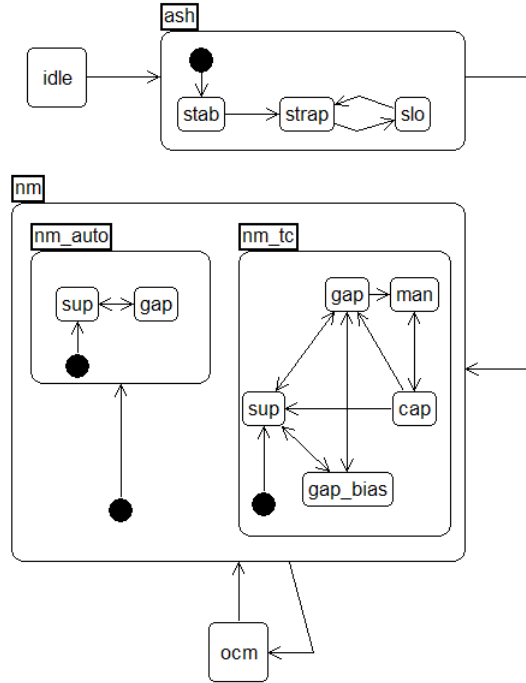


Figure 2. Hierarchy of states of the AOCS

refined, which turns to defining mode dependent behaviors. A part of the AOCS state machine is illustrated by Figure 3.

```

system implementation AOCS_FUNCTION.N0
  modes
    IDLE: initial mode; ASHMODE: mode;
    NLMODE: mode; OCMLMODE: mode;
  annex behavior_specification {**
    mode transitions
      IDLE -[TC.ASH_REQ?]-> ASHMODE;
      NLMODE -[TC.OCML_REQ?]-> OCMLMODE;
      ASHMODE, OCMLMODE -[TC.NM_REQ?]-> NLMODE;
      ASHMODE, NLMODE, OCMLMODE -[TC.IDLE_REQ?]-> IDLE;

    composite mode ASHMODE
      states
        STAB: initial state;
        STRAP, SLO: state;
      transitions
        STAB -[on Bdot_Control_Law_Converged]-> STRAP;
        STRAP -[on Sun_Presence]-> SLO;
        SLO -[on Sun_Presence = false]-> STRAP;
      end ASHMODE;
    ...
  **};
end AOCS_FUNCTION.N0;

```

Figure 3. AOCS state machine

Although the Behavioral Annex is projected to be used in thread and subprogram implementations, it can also be used in system implementations, when these need to have their behavior represented with enhanced state machines.

5.2. Dynamic architecture (N1)

The model described at level N1 details the synchronization of threads that access the 1553 bus[5]. It is used to perform schedulability analysis. The bus is here used in a very deterministic way: each thread has a periodic access to the bus during predefined time windows. Consequently, the bus is modeled (Figure 4) by a periodic thread sending interrupts when data can be accessed by designated threads. In order to provide precise synchronization points, the bus is modeled using concurrent statemachines, each of them waiting for a given delay before sending a signal specifying that some data has been transferred. This model defines a simulation of the bus behavior and does not reflect the actual physical architecture.

```

thread avb_bus
features
  int_it: out event port;
  eof_it: out event port;
properties
  Dispatch_Protocol => Periodic;
  Period => 125ms;
end avb_bus;

thread implementation avb_bus.i
annex behavior_specification {**
states
  s0: initial concurrent state;
  s1: complete join state;
transitions
  s0  $\xrightarrow{!}$  s1;
concurrent state s0
composite state t1
  states
    t1i: initial exit state;
  transitions
    t1i  $\xrightarrow{!}$  t1i { delay(38ms); int_it!; };
end t1;
composite state t2
  states
    t2i: initial exit state;
  transitions
    t2i  $\xrightarrow{!}$  t2i { delay(79ms); int_it!; };
end t2;
  ...
end s0;
**};
end avb_bus.i;

```

Figure 4. The N1 Bus Model

Components connected to the bus are modeled using bus handlers and applicative tasks. They were initially considered as periodic [9], but we propose here a sporadic point of view which is closer to the event driven behavior of the tasks and better matches the AADL model. The bus handler (Figure 5) is awakened by the timer at each period, performs initialisations if needed and then waits for the interrupts sent

by the bus. Acknowledgements are sent to the application layer each time data is received.

```

thread AVB_HDLR
features
  timer: in event port;
  int_it: in event port;
  eof_it: in event port;
  aocs_ack: out event port;
  dor_ack: out event port;
properties
  Dispatch_Protocol => Sporadic;
  Period => 10ms;
end AVB_HDLR;

thread implementation AVB_HDLR.i
annex behavior_specification {**
states
  s0: initial complete state;
  s1, s2, s3: complete state;
transitions
  s0  $\xrightarrow{!}$  timer;
  s1  $\xrightarrow{!}$  int_it;
  s2  $\xrightarrow{!}$  aocs_ack;
  s3  $\xrightarrow{!}$  dor_ack;
  s0  $\xrightarrow{!}$  s1;
  s1  $\xrightarrow{!}$  s2 { aocs_ack!; };
  s2  $\xrightarrow{!}$  s3 { dor_ack!; };
  s3  $\xrightarrow{!}$  s0;
**};
end AVB_HDLR.i;

```

Figure 5. The N1 Bus Handler Model

The third layer consists in applicative threads. They wait for data sent by several bus handlers before processing them. This multiple synchronization mechanism is not directly supported by the AADL execution model. Transitions fired by the received events can count the number of received messages until the expected number of data is obtained. However, this pattern is often used in the considered case study. AADL and the annex should offer a construct to directly support this mechanism.

5.3. Feedbacks

Timed behavior For the model to be correct, the bus thread must be dispatched at the start of the period and more generally, actions must be performed as soon as possible. However, this maximal progress semantics is not strong enough: delays must be strictly enforced. It corresponds to the timed transition system semantics [3] where a time interval is associated to a transition. If the transition has been active for at least the minimum bound of the interval, it must be fired before the maximum bound. Checking the model at this level needs a schedulability analyser that takes into account timed behaviors, as for example the Tina tool[1].

Sporadic threads The case study illustrates the need to specify more precisely the dates of dispatch of a sporadic thread. They appear to be irregular (three times with a given

period) but well known. So dispatch dates should be synthesized by an analysis tool taking into account the global behavior of the system, or specified in order to perform modular analysis. In the latter case, a list of dispatch dates is not sufficient because they must be associated to events in order to simulate the behavior of the thread. In fact, the specification of the environment is needed. It can be defined by another AADL thread with having a complementary interface.

Multiple synchronizations Synchronization on a conjunction of events is supported by several real time operating systems. For example RTEMS[4] offers the primitive `rtems_event_receive` which suspends a thread until all or some of a set of conditions are satisfied. Such a call could be added to the AADL API. Such an extension is already needed to support the dispatch of a thread waiting for one of the events in guards of transitions starting from the current state. This new feature would allow to express transitions with guards containing several receipts. The combination of the two needs (or-wait, and-wait) leads us to the proposal of an advanced dispatch condition expressed in AADL using a boolean condition over ports.

As an example, the applicative layer could be specified as in Figure 6 where the guard of the transition starting from the ACC state contains two events, which means that the thread is dispatched when the two event ports have received a signal. This extension avoids manual encoding of the mechanism either through several sequences of signal reads or using counters.

```

thread implementation PL_CYC.i
annex behavior_specification {**
states
  RQ: initial state;
  wait_PLB, ACC, DA: complete state;
transitions
  RQ  $\neg$ {start}  $\rightarrow$  wait_PLB { computation(2ms); };
  wait_PLB  $\neg$ {PLB_ACK}  $\rightarrow$  ACC { computation(4ms); };
  ACC  $\neg$ {PF_END? & AOCs_END?}  $\rightarrow$  DA { ... };
**};
end PL_CYC.i;

```

Figure 6. The N1 Applicative layer

6. Conclusion

In this paper, we have presented the main features of AADL and its behavioral annex. The annex has been validated through a reengineering of a flight software design realized in the ArchiDyn project. The original model has been designed to perform real-time analysis but suffered from insufficiencies of the AADL language to express behavioral information. The annex overcomes these problems

and provide a model which is close to the actual architecture of the considered system. The study also enlightens some possible enhancements of both the annex and the AADL execution model: new synchronization primitives should be added to the AADL runtime. Concerning the annex, the next steps are related to its support by editing and analysis tools, which is at that time limited to parsing and static verification. Advanced model checking also needs the ability to express real-time properties, which could be done either through the AADL property mechanism or through the definition of a new annex.

References

- [1] B. Berthomieu, F. Peres, and F. Vernadat. Bridging the gap between timed automata and bounded time petri nets. In *Proceedings of FORMATS 2006*, number 4202 in LNCS. Springer Verlag, 2006.
- [2] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis & design language (aadl): An introduction. Technical report, CMU, 2006. <http://www.sei.cmu.edu/publications/documents/06.reports/06tn011.html>.
- [3] T. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for real-time systems. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 353–366, New York, NY, USA, 1991. ACM Press.
- [4] RTEMS. Rtems c user's guide. http://www.rtems.com/onlinedocs/releases/rtmsdocs-4.6.99.3/share/rtems/pdf/c_user.pdf.
- [5] i. SBS Technologies. An interpretation of mil-std-1553b. http://www-corot.obspm.fr/COROT-ETC/Files/1553_overview.pdf.
- [6] SEI. the sei open source aadl tool environment (osate). <http://www.aadl.info/>.
- [7] F. Singhoff. Cheddar release 2.x user's guide. Technical Report singhoff-01-2007, LISYC, Feb 2007.
- [8] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Scheduling and memory requirements analysis with aadl. *Ada Lett.*, XXV(4):1–10, 2005.
- [9] D. Thomas, L. Planche, and L. Mailet. Architecture dynamique des logiciels enfouis:modélisation d'un logiciel de vol. Technical report, CNES, 2006. http://cct.cnes.fr/cct05/public/2006/seminaires/archi.dyna.vol/modelisation_PLEIADES.pdf.
- [10] TopCased. Toolkit in Open-source for Critical Applications & SysEms Development. <http://www.topcased.org>.
- [11] S. Vestal. Formalizing avionics software architectures. In *journées Systèmes et Logiciels Critiques*, Grenoble, 2001. Verimag. <http://www.systemes-critiques.org/journees2001.php>.