Towards a formal semantics for AADL execution model

Jean-François Rolland, Jean-Paul Bodeveix, David Chemouil, M Filali, Dave Thomas

▶ **To cite this version:**

Jean-François Rolland, Jean-Paul Bodeveix, David Chemouil, M Filali, Dave Thomas. Towards a formal semantics for AADL execution model. 4th European Congress on Embedded Real Time Software and Systems (ERTS 2008), 3AF : Association Aéronautique et Astronautique de France; SEE : Société de l'électricité, de l'électronique et des technologies de l'information et de la communication, Jan 2008, Toulouse, France. hal-02269846

HAL Id: hal-02269846
https://hal.science/hal-02269846

Submitted on 23 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Towards a formal semantics for AADL execution model

J-F. Rolland[1*], J-P Bodeveix[1], D. Chemouil[2], M. Filali[1], D. Thomas[3]

1: IRIT, CNRS, Université Paul Sabatier, 118 Route de Narbonne, Toulouse
{rolland,bodeveix,filali}@irit.fr
2: CNES, 18avenue Edouard Belin, Toulouse
3: EADS ASTRIUM, 31 avenue des Cosmonautes, Toulouse
dave.thomas@astrium.eads.net

Abstract: In this paper, we present a specification in TLA+ of an AADL execution model. This formal specification is used for deriving a prototype verification tool for AADL within the TOPCASED development environment.

Keywords: architecture description languages, dynamic properties, formal semantics, model checking

1. Introduction

Model driven engineering has put forward a set of techniques and tools to enhance the production of reliable software. In this context, architecture description languages are now well accepted as a way to express the relevant dynamic properties that one must first specify and then ensure. Since, we are at the model level, in order to specify and ensure, in a sound way, these dynamic properties, we must rely on a well defined execution semantics. Our work is related to that topic. More precisely, we are interested in enhancing the precise semantics of the AADL execution model by a formal semantics. More precisely, we look for specifying formally relevant fragments of the AADL execution model. For this purpose, we have used the TLA+ [5] language. TLA+, the Temporal logic of Actions is well suited for describing, in an abstract way, the behavior of a system. Actually, TLA+ has already been used to specify as well hardware protocols, e.g., memory protocols, as software protocols, e.g., distributed consensus protocols.

The rest of this paper is organized as follows: Section 2 introduces AADL and the features we are interested in. Section 3 presents the main features of our formal model. Section 4 presents the prototype that we have elaborated. Section 5 discusses our perspectives with respect to our formal model and to our tool. Section 6 draws some conclusions.

2. AADL

AADL [1] is an architecture design language standardized by the SAE. This language has been created to be used in the development of real time and embedded systems. As a successor of MetaH [8], AADL capitalizes more than 10 years of experiments. MetaH is a language developed by Honeywell Labs and used in numerous experiments in avionics, flight control, and robotic applications. AADL also benefits from the knowledge on ADLs acquired at CMU during the development of several ADLs, like ACME[6] and Wright[7].

2.1 The language

AADL includes all the standard concepts of any ADL: components, connectors used to describe the interface of components, and connections used to link components. The set of AADL's components can be divided in three

*Work funded by CNES and EADS Astrium Satellites

partitions, the software components (process, thread, thread group, subprogram, and data), the hardware components (processor, bus, memory, device), and a System component. Components can communicate through ports, synchronous calls, and shared data. A process represents a virtual address space, or a partition, this address space includes the program defined by its sub-components. A process must contain at least one thread or thread group. A thread group is a logical organization of threads in a process. A thread represents a sequential flow of execution, it's the only AADL component that can be scheduled. A subprogram represents a piece of code that can be called by a thread or another program. A data models a static variable used in the code, they can be shared by threads or processes.

A processor is an abstraction of the hardware and the software in charge of the scheduling and the execution of threads. The memory represents any platform component that stores data or binary code. The buses are communication channels used to connect different hardware components. The devices represent interfaces between the system described and its environment.

Systems allow to compose software components with hardware components. The interactions can be defined at a logical and a physical level. At a physical level, software components are associated to hardwares component, a thread to a processor, or a data to a memory for example. The logical level is used to describe the communication between hardware and software. At a logical level we can define communication connections between processors or devices and software components.

AADL uses the notion of mode to determine a set of active components. This mechanism allows to describe dynamic architectures. The set of active components can be modified by the reception of an event.

The AADL standard describes a strict semantics of execution, this semantics is customizable using properties. We will present only a subset of AADL. We don't take into account the hardware components. Modes are not modeled yet, but it is planned to integrate them in our model. We will present this semantic aspect for the communication through ports, the scheduling and the communication through shared data.

2.2 Communication through ports

AADL proposes three types of ports: data, event and event data ports. A port is declared to be in an input, output or input/output mode. It can be used to transmit data or control or both. Ports are used to describe the interface of a component. Data transmitted through ports is typed. Each input port has a fresh variable to define the state of the port, if a port has not received anything between two thread dispatches this variable is set to false. A buffer is also associated with each input port, when an output port sends a data or an event it modifies these buffers. On the dispatch of a thread these buffers are copied into the local memory of the thread. Some properties permit to customize the behavior of event and event data ports. The property "Queue_size" determines the maximum number of events that can be received. “Overflow_handling_protocol” describes the behavior of the port in case of overflow, the two default politics are drop newest and drop oldest. The "Dequeue_protocol" describe the way elements in the queue are accessed, one by one ("OneItem") or all at once (“AllItems"). Data ports have the simplest behavior, data is sent at the end of the thread's execution and is received at the next dispatch of the receiving thread. Event and event data ports have a very close behavior, they can send an event or event data anytime during the

execution of a thread. Events or events data sent are queued in the destinations ports. Input event and event data ports are delivered at the dispatch of the thread. For periodic threads that are harmonic, a data connection can be declared as immediate or delayed. If the connection is delayed data is sent at the end of the period of the sending thread. If the connection is immediate the receiving thread must wait the sending thread to complete and it receives data at the start of its execution.

2.3 Communication through shared variables

As all AADL components, data has a type and an implementation. The internal structure of the data is described in the data implementation. We can specify that different components have a shared access to a data subcomponent using the “require data access” connector. The “provide data access” connector is used to represent that a component allows other components to access to one of its data subcomponent. The concurrency protocol used to access to a data is defined by a data property called “concurrency_control_protocol”.This concurrency protocol can be implemented through different concurrency control mechanisms such as mutex, semaphore... The data is locked when the thread enters in a critical region, i.e. when the thread accesses to the data. But the AADL standard does not allow to describe precisely when the data is accessed. The “provide” and “required data access” connectors have a “Provide_Access” and a “Required_Access” properties used to defined the different form of access needed or provided (read only, write only, read write).

2.4 Scheduling strategy

Thread models: Threads are the only components that have an execution semantics. AADL supports the classic types of dispatch protocols, a thread can be declared as periodic, aperiodic, sporadic or background. All the standard properties (WCET, deadline,...) used to described a real-time system exist in AADL. Threads have two predeclared event ports : dispatch and complete. The dispatch port is used for aperiodic or sporadic threads. If this port is connected all other ports of the thread do not trigger the dispatch. It's a very common behavior for an aperiodic or a sporadic thread to send an event on completion. In AADL, we do not specify when an event is sent. The complete event ports used to send an event at the end of the execution.

Basic scheduling strategy: All the thread have the same life cycle, this cycle can be represented as an automaton. All threads start in the awaiting dispatch
state. The dispatch condition depends on the thread's type. If the thread is periodic it will be dispatched at every period. At this time, delivery occurs for all its input ports. An aperiodic or a sporadic thread that does not have its dispatch ports connected is dispatched each time it receives an event. Delivery occurs only for the port that triggers the dispatch and the data ports. If its dispatch port is connected, it is dispatched each time it receives an event on this port, and delivery occurs for all its others ports. The thread in the active state that has the maximum priority starts or continues its execution. The priority of the thread is determined by the chosen scheduling policy (RMA, EDF, LLF). This policy is specified by a property of the model. When a thread is dispatched it can have a higher priority than the executing thread. In this case, the executing thread is preempted and goes back to the active state. When a thread ends its execution it goes to the “awaiting_dispatch” state until its next dispatch. At this time, all the output data ports of the thread are read and their content is sent to their respective destination ports.

Impact of shared data on scheduling: The precedent behavior is slightly modified when

we used shared variables with concurrency control. In order to take into account the shared variables we just have to add a state to the automaton. When an executing thread tries to access to a locked shared variable, it goes to this state. It can go back to the active state when the variable is released. Here, we do not specify when the data is locked. It depends on the implementation used. If the implementation describes the behavior of the thread in a very precise way, you can lock the shared variable for a very short time, just when it is accessed. But if the model describes a very abstract behavior, the most strict implementation is to lock the shared variable when the thread starts its execution and to unlock it at the end of the execution.

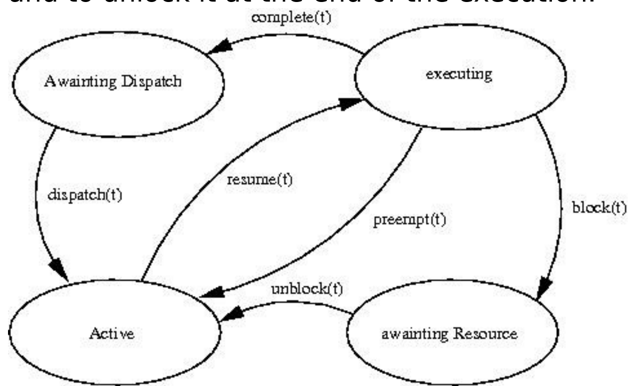

Figure 1: Thread's life cycle

3. A formal model for AADL execution model

In this section, we are concerned by setting a formal semantics for the AADL execution model. Although, AADL brings precise semantics for real time components, to the best of our knowledge, such semantics has not been formalized with a formal notation yet. The goal of such a semantics can be twofold:

- first it can be used to reason about an AADL design formally. Actually, since our semantics is stated in the TLA formalism, it will be possible to perform some properties verification through model checking.
- second it can be used as a formal specification for the development of an AADL execution platform. One can imagine that an actual implementation would be certified with respect to the proposed model.

We are concerned by a subset of the execution model only, we try to define a subset small enough to be formalized easily but with enough expressiveness to perform small tests. The only components used in our model are threads and data. The communication between threads can be done through ports or shared variables. For the scheduling, we consider only periodic and aperiodic threads. We implement a fixed priority policy for the scheduling, with preemption, and a simple access control protocol for shared variables.

3.1 A brief presentation of TLA+

Specification in TLA+: TLA+[5] specifications are organized into modules. A module contains constants, variables, assumptions and definitions.

We are concerned with transition systems. While their state spaces can be defined using variables with values in sets as just given, TLA+ definitions are used to introduce the following:

- The set of initial states, using a predicate usually called *Init*.
- The set of transitions, using action predicates. An action is a formula containing primed (next state) variables and unprimed (current state) variables. Such a formula describes the relation between the current state and next state values of the variables.

Time in TLA+: In this section we present a way of representing the evolution of time and the expression of time constraints. As TLA does not have pre-defined constructions to manipulate time, we use an explicit time approach proposed by Lamport. The basic principle used by Lamport is obvious, we add a variable called “now”. The evolution of this variable represents the evolution of time. This

variable is manipulated trough an operation tick, this operation increases the value of “now”. In order to express time constraints we can use three kind of timers:

- expiration timer: The tick operation does not change the value of the timer. It is set to a value greater than now and the timeout occurs when now = timer.
- count down timer: The tick decrease the value of the timer. The timeout occurs when timer = 0.
- count up timer: The tick increase the value of the timer. The timeout occurs when the timer equals a predefined constant.

Timers can be set up in the tick operation, or in other part of the next transition.

3.2 General architecture

We have developed a generic TLA architecture easily customizable. The kernel and ports modules model the behavior of the execution model described in the AADL standard. The “threads behavior” module contains the behavior of each thread. This behavior is represented be a simple relation between the input of a thread and it's output. We consider that the calculation is atomic, even if the thread can be preempted. The AADL model is a set theory representation of an AADL model. All the threads, ports, shared variables are represented by sets, the interface of a thread is defined by relations (associations between ports, shared variables and threads). The properties are also represented by relations. The mapping between an AADL model and this configuration module is really easy and can be done automatically. The kernel module contains the representation of the thread's life cycle, and shared variables mechanisms. We model in this module all the scheduling. For each type of ports we have a corresponding module in TLA. Each of these modules represent all the ports of its type, for example the out data port module represents all the output data ports of the models. These modules are parameterized by the sets and relations defined in the module representing the AADL model. Another way to represent ports would have been to create one module for each port of the model but the generation from an AADL model would have been harder.

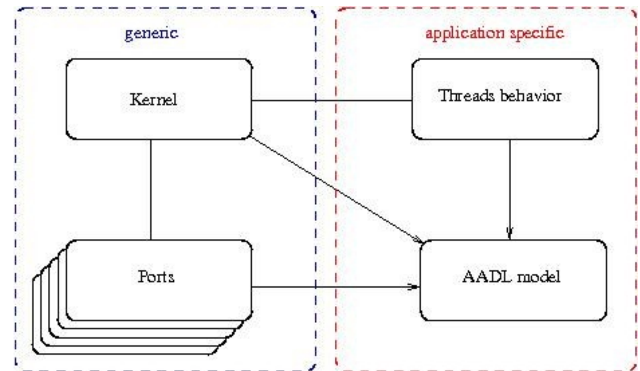

Figure 2: Global structure of our TLA specification

3.3 AADL ports

The structure of our application has an impact on the conception of operations. We don't have simple variables, all the variables are functions from a set of ports into a set of data, or naturals. For example a simple event counter is function from the sets of input event ports. When we modify those variables we have to calculate the relation that associate modified ports to their new values and then modifies the variable according to this relation.

All the input ports have the same structure, a set of variables, a set of constants and an operation. The variables used are a buffer, filled by the input ports, a delivered variable and a fresh variable accessible from a thread. Each input module contains a set of all its ports, some additional constant relations describe the properties associated to a port. The deliver operation describes how the elements are copied from the buffer to the delivered variable.

Similarly all the output ports have a very close structure. Each output port module has two sets to define the input and output ports, a variable for the connections between ports, and some additional relations to describe the

properties associated to a port. A “store” or “raise_event” operation describes the behavior of the port.

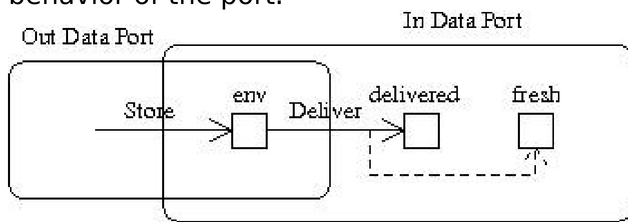

Figure 3: Structure of data ports

Not all the AADL specification is modeled. We don't use immediate or delayed connexion between ports, we use only the drop newest policy for event and event data ports. We just send event and event data at the end of the execution. We could specify that an event can be sent at any time but it would led to a model on which we could not do any verification, the number of generated traces being too big.

3.4 AADL Threads

Threads are the only elements of AADL with an execution semantics, the module corresponding to threads is the center of our architecture. All the system's variables are declared in this module, all the instantiation of other modules are also done here. Threads are represented by a set, the interface and the different properties of the threads are relations.

As for ports, we don't respect totally the AADL standard. Currently we use only periodic and aperiodic threads. For aperiodic threads we support only one type of dispatch. We consider that the behavior of the thread is a simple relation between its input and its output. Those relations are described in a separate TLA module.

The principle is obvious, we just have to encode the automaton described in the first section. A state of the figure corresponds to one subset of the “Thread” set. Each transition corresponds to a TLA operation, the evolution of the whole system is a disjunction of these operations. We have to ensure that transition are done in a certain order. In accordance with the technique presented by Lamport, we use a global variable to represent time, and timers to model different protocols of scheduling. For example all threads have a deadline timer, initialized at the dispatch to the value of the deadline property of the thread. This timer is decreased on each clock tick if the thread is active or executing. If it becomes less than zero the deadline is missed. The whole system acts as a stopwatch automaton, transitions are guarded by timers and these timers are decreased only in certain states.

4. Prototype

4.1 Framework

In this section, we outline the different tools our framework relies on. For each one, we give its main features.

Osate: OSATE[9], Open Source AADL Tool Environment, is an Eclipse[14] plugin dedicated to the edition of AADL models. The metamodel of AADL is described in EMF, the Eclipse language for metamodels. This tool provides the backend for manipulate AADL models in text or XMI. Moreover it includes some analysis tools.

Topcased: The TOPCASED [3] project is concerned by the definition and the implementation of an Open Source Environment for the development of Critical Applications. With respect to development TOPCASED supports the so called model driven engineering. Actually, modeling notations like UML, AADL, SYSML and SDL are currently supported by the TOPCASED toolkit. The architecture of TOPCASED is illustrated by the following figure:

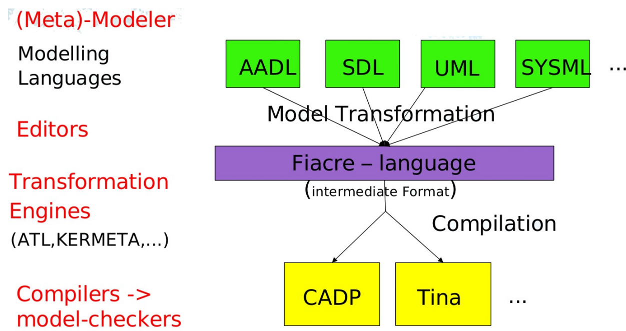

Figure 4: TOPCASED architecture

One of the features of TOPCASED is to promote the use of the so-called pivot languages. The following table illustrates some of the tools currently available in TOPCASED and the corresponding pivot language.

Purpose	Pivot language	Tool
Data modeling and transformation	ecore	ATL[15], Kermeta[17], Acceleo
Verification	Fiacre	Fiacre engine

Acceleo: Acceleo[10] is an open source code generator. As it is an eclipse plugin, it uses metamodels described in EMF. Even if it's main usage is to generate code from UML models, it accepts other metamodels, notably AADL metamodel. From our point of view, one major advantage of Acceleo is that it allows to define Java services to be executed on nodes of XMI tree. This permit to call OSATE built in methods to recover information. For example, the period of a thread can be defined as a property of the thread group, of the thread, of the thread implementation... OSATE supplies Java methods that finds this kind of information wherever it is defined. At last Accelleo is now part of TOPCASED.

TLA tools: TLA tools are open source. They consist in:

– a syntactic analyzer;
– a LaTeX pretty printer ;
– a model checker and a simulator for a subset of TLA.

Currently, the verification process is supported by the TLC model checker. The proof process is not currently supported as such. In fact, the proof process can be considered as supported in the cases where the exhaustive exploration of the model is possible.

Architecture of the prototype: We use Acceleo to define templates that express the relation between an AADL model and it's representation in TLA. We have one template for the generation of the architecture part of the model, plus non functional properties (the AADL_model module), and a template for the generation of the TLA module that contains the behavior of threads. After editing an AADL model we can generate TLA modules by applying these templates. The generated modules are used to parametrize our TLA specification of the AADL execution model. We can then run the TLC model checker to verify some properties on the model. In the next parts, we will show what kind of models and what kind of properties can be checked.

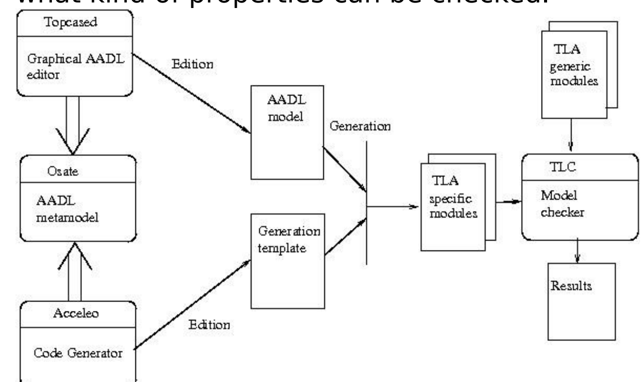

Figure 5: architecture of the application

4.2 Restrictions on models

Here we have to deal, with two types of restrictions, those that come from our representation of the execution model, and those that come from our translator. In the latter case, there is mainly some syntactic problems. In the translator we don't take care about name-spaces, but in the TLA

representation, we can't have, for example two threads with the same name. In AADL, nothing forbids two threads to have the same name if they are not in the same container (same process for example). This naming problems occurs for all AADL elements. Thus we have to take care of the different name we use in a model. This restriction can be easily circumvented by adding to each element name the name of its container. As this work is done on the instantiation of an AADL model, the other solution would be to base our translation on the instance of the AADL model.

The second type of restrictions comes from our decision to use only a subset of AADL in the TLA specification. Here we will try to list the major limitations of our model. As we said in the first part of the paper, we consider that threads are the only elements that have an execution semantics. The communication between threads can be made through ports or shared variable. For the thread we need to give all the needed informations for the scheduling (period, wcet, deadline). A shared variable is represented as a data component accessed by threads. Each thread that access to this variable must have a requires_data_access port, type of access to the data is defined in the properties of this port. We consider that the thread lock the data at the beginning of its execution and release the lock at the completion time. For each event or event data port we define the length of the queue. The communication between threads only happens at the dispatch time and at the completion time.

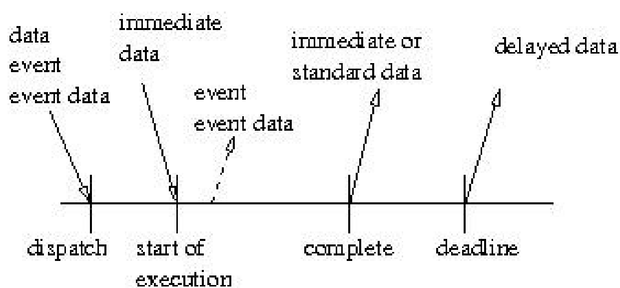

Figure 6: Timing of communication in AADL

We implement a small part of the thread life cycle, as defined in the first part of the paper. We don't take into account the errors, activation or deactivation mechanisms.
Currently the thread behavior generator is a simple translator. It generates a standard behavior: at the end of the execution the thread send a data on each data and event data ports and emit an event on each port.

4.3 Properties

In the current version of the prototype we check for three type of properties, the schedulability of the system, the size of buffers, and the protection of shared data. These three type of properties are verified by the model checker. The shedulability analysis takes into account periodic and aperiodic threads. Aperiodic threads are dispatched on the reception of an event. We considers that there are two type of shared variables, protected or not protected. In the first case a thread can access to the variable only if it is not already in use. In the latter case we consider that the scheduling guarantee the integrity of the data. When a property is violated, the model checker gives the trace of an execution trace that leads to the violation of the property.

5. Perspectives

In this section we describe the current and future work performed around AADL and the framework presented here.

5.1 Modes

We started to study the mode mechanisms described in the AADL standard. A first paper will be published in [11]. In this paper we describe the behavior of a system during a mode switch. This description has been done in TLA+. We try to study precisely all kind of mechanisms that can be part of the mode switch. As this is a very abstract vision of mode switch, the aim of this initial description

is to capture the semantics of mode change; verifications should be possible once we have refined this initial description by concrete implementation of mode switching. Currently, we have begun to sketch Giotto and AADL mode switching. . Our goal is to integrate this work in the framework we have presented here. This would give us the possibility of checking timed properties on the mode switch, for example we could check that a mode switch must happen in less than a particular time. It is interesting to remark that mode mechanisms in asynchronous systems requires more attention than in synchronous systems [12],[13]; actually, since we do not assume the basic hypothesis of the synchronous approach: zero time computation, deterministic concurrency and instantaneous communication, we have to handle the transitional aspects related to these concepts. From our point of view, the formal specification of these aspects is challenging and is worth consideration.

5.2 Releasing constraints on communications

In the presented work we impose strong constraints on communications, i.e. we allow communications only on the dispatch and on the complete. Those restrictions are useful for the model checking but. Fortunately AADL version 2 will introduce special properties for defining more precisely the timing of communications. By integrating those properties in our models we will be able to describe some interactions between threads during their executions. The same type of technique can be applied for defining more precisely the instants where a share data is accessed.

5.3 Generating thread behavior

In this work the behavior of threads can not be parameterized. An AADL extension exists to define the behavior of threads, it is the behavioral annex[16]. We should use this language as an entry point for our generator to derivate the behavior of threads in TLA. We have already done several experimentations for the definition of the semantics of the behavioral annex in TLA. It follows that the integration within our framework should be straightforward.

5.4 Evolution of the prototype

The translation schemes defined in our prototype are very simple and must be detailed and validated. Currently the edition of an AADL model an the translation into TLA modules can be made in the TOPCASED environment. We also want to integrate the process of model checking to this environment in order to have a single integrated tool.

5.6 Scalability

At this time we only try this prototype on small examples. With some realistic models the generation of TLA modules should work well. But the model checking of real models with TLC might be too long. We have to test our prototype in such case and possibly choose another model checker.

6. Conclusion

In this paper, we have presented our current work concerning the formalization of the AADL execution model. This work has made us much more confident about the understanding of the basic AADL execution model mechanisms. We have also related our first experiments on the use of the transformation tool Acceleo. Aside, from the perspectives given in the preceding section, we should also mention that our work is currently the starting point for the translation from AADL to Fiacre: the verification pivot language of TOPCASED. Moreover, our work has also been used as the starting point for the translation[19] between a subset of AADL and the real time specification for Java: RTSJ[18].

Aknowledgements: We would like to thank Peter Feiler for initial discussions about this work.

7. References

[1] SAE Aerospace: "SAE AS5506 : Architecture Analysis and Design Language (AADL)", SAE Internatinal, 2004.

[2] Jean-Paul Bodeveix, Mamoun Filali and Jean-François Rolland: "AADL execution model semantics, AADL communication semantics in TLA", Technical Report, IRIT, 2007.

[3] Topcased: "Toolkit in OPen-source for Critical Apllications and SystEms Development", http://www.topcased.org .

[4] Jean-Raymond Abrial: “The B Book: Assigning programs to meanings”, Cambridge University Press, 1996.

[5] Leslie Lamport: “Specifying Systems:The TLA+ Language and Tools for Hardware and Software Engineers”, Addison-Wesley, 2002.

[6] D. Garlan and R. Monroe and D. Wile: “ACME: An Architecture Description Interchange Language”, CASCON'97, Toronto, 1997.

[7] Robert Allen: “A Formal Approach to Software Architecture”, Carnegie Mellon, School of Computer Science, 1997.

[8] S. Vestal: “MetaH User's Manual”, Honewell Technology Drive, 1998.

[9] The SEI AADL Team: “An Extensible Open Source AADL Tool Environment (OSATE)”,Software Engineering Institute, 2006.

[10] Acceleo: “An open source code generator”, http://www.acceleo.org/pages/home/en

[11] Jean-François Rolland, Jean-Paul Bodeveix, Mamoun Filali, Dave Thomas and David Chemouil: “Modes in asynchronous systems”, UML&AADL'08, IEEE ICCECS, Belfast (to appear).

[12] Florence Maraninchi and Yann Rémond: “Mode-Automata: a new domain-specific construct for the development of safe critical systems”, Sci. Comput. Program., 2003.

[13] Jean-Pierre Talpin, Christian Brunette, Thierry Gautier and Abdoulaye Gamatié: “Polychronous mode automata”, EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software, Seoul, 2006.

[14] Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick, and Timothy Grose: “Eclipse Modeling Framework”, Addison-Wesley, 2003

[15] Jean Bézin,Erwan Breton,Grégoire Dupé and Patrick Valduriez: “The ATL Transformation-based Model Management Framwork”, IRIN, 2003

[16] Ricardo~Bedin França, Jean-Paul Bodeveix, Mamoun Filali, Jean-François Rolland,David Chemouil and Dave Thomas. The AADL behaviour annex experiments and roadmap.12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007), Auckland, New Zealand, 11/07/07-14/07/07, pages 377—382, http://www.computer.org, 2007, IEEE Computer Society.

[17] Zoé Drey, Cyril Faucher, Franck Fleurey, and Didier Vojtisek: “Kermeta language reference manual”, IRISA, 2006.

[18] A.Wellings. Concurrent and Real-Time Programming in Java. Wiley, 2004.

[19] Jean-Paul Bodeveix, Raphael Cavallero, David Chemouil, Mamoun Filali and *Jean-François* Rolland. A mapping from AADL to Java-RTSJ.International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES), Vienna, Austria, 26/09/07-28/09/07}}, ACM International Conference Proceeding Series, pages 165—174.

8. Glossary

AADL: Architecture Analysis & Design Language

ADL: Architecture Design Language

ATL: Atlas Transformation Language

CMU: Carnegie Mellon Universitie

EMF: Eclipse modeling Framework

OSATE: Open Source AADL Tool Environment

TOPCASED: Toolkit in Open Source for Critical Applications & Systems Development

UML: Unified Modeling Language

SEI: Software Engineering Institue

XMI: XML Metadata Interchange