Formal Verification of AADL models with Fiacre and Tina

B. Berthomieu*‡, J.-P. Bodeveix†‡, S. Dal Zilio*‡, P. Dissaux§, M. Filali†‡,
P. Gaufillet¶, S. Heim‖, F. Vernadat*‡
*CNRS ; LAAS ; 7 avenue colonel Roche, F-31077 Toulouse, France
†CNRS ; IRIT ; Université de Toulouse, 118 route de Narbonne, F-31062 Toulouse, France
‡Université de Toulouse ; UPS, INSA, INP, ISAE, UT1, UTM ; F-31062 Toulouse, France
§Ellidiss Technologies; 24 quai de la douane, 29200 Brest, France
¶AIRBUS Operation S.A.S.; 316, route de Bayonne F-31060 Toulouse, France
‖CS ; ZAC de la Grande Plaine - Rue Brindejonc des Moulinais, F-31506 Toulouse, France

Abstract: This paper details works undertaken in the scope of the Spices project concerning the behavioral verification of AADL models. We give a high-level view of the tools involved and describe the successive transformations performed by our verification process. We also report on an experiment carried out in order to evaluate our framework and give the first experimental results obtained on real-size models. This demonstrator models a network protocol in charge of data communications between an airplane and ground stations. From this study we draw a set of conclusions about the integration of model-checking tools in an industrial development process.

Keywords: Architecture Description; Verification and Validation; Model-Driven Engineering; Transportation.

1. Introduction

As in many applicative domains related to Information Technology, the size and complexity of avionic applications is constantly growing. It is now common to embed hundreds of mega bytes of code and data in on-board computers, and applications typically include dozens of heterogeneous, loosely connected features. This complexity explosion led the avionic industry to rely on new architectures and operating systems, more powerful, but also more complex than their ancestors. While these new architectures make development and maintenance easier, it is more difficult to fully understand, analyze and test these systems. This trend is a powerful incentive to improve model-based development techniques and to bring architecture description languages such as AADL — the SAE Architecture Analysis and Design Language — on the designer's desktop

In order to support model-based development, companies from the French Aeronautics, Space and Embedded Systems competitivity pole (Aerospace Valley) have joined their efforts to develop a common set of methods and tools. The goal is to deliver an industrial strength system/software development platform for embedded systems. The Topcased [9] initiative is part of this effort and AADL is among the first languages supported in this project. Topcased is also the name of a toolkit based on the Eclipse platform and concepts that provides an open source, model oriented set of tooling and standard implementations.

AADL is an architecture description language that allows to describe both the hardware and software components of a system. A key extension to this standard is the addition of a Behavioral Annex for describing more precisely thread activities. Today, static semantic verification of models — architectural or not — is well known and commonly used, but this is not the case with behavioral verifications, especially of architectural models. In this paper, we describe a formal verification toolchain for AADL enriched with its behavioral annex. This toolchain (see Fig. 1), is connected for its input to ADELE [1], a semantic editor for the elaboration of AADL models. At the other end, verification activities ultimately relies on the Tina toolset. In-between, the generation of Tina models from an AADL description relies on the Fiacre formal specification language [8].

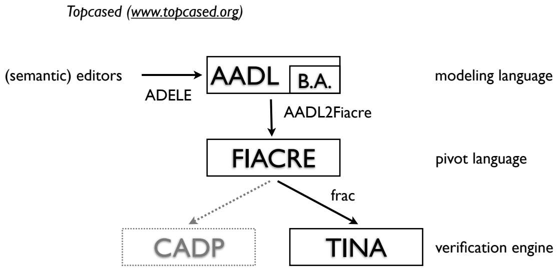

Figure 1. AADL to Tina toolchain

The paper details works undertaken within the ITEA2 Spices project concerning the behavioral verification of AADL models. In Sections 2 and 3 we give a high-level view of the tools and languages involved and illustrate the successive transformations required by our verification process. In Section 4, we report on the first representative experiment carried out in order to evaluate our framework. This demonstrator models a network protocol in charge of data communications between an airplane and ground stations. More precisely, we models the dynamic architecture of a software component implementing the onboard part of this protocol. From this study, we draw a set of conclusions about the integration of model-checking tools in an industrial development process.

2. Modeling Dynamic Architectures with AADL and its Behavioural Annex

In the scope of the SPICES project, Airbus has chosen AADL for modeling the dynamic architecture of real time softwares. A key point of AADL is to enable the precise description of both the software components of a system: process, thread, data, . . . , as well as the execution platform supporting them: processor, device, bus, memory, . . . , by using detailed properties for each component. The language includes features (used to describe the interface of components), and connections (used to link components). Components can communicate through ports, synchronous calls, and shared data. The AADL execution model is suitable to describe real-time systems because it includes the main types of dispatch protocols for threads (periodic, aperiodic, sporadic, background) and the standard scheduling properties (period, priority, deadline, WCET, scheduling policy, . . .).

The expressivity of AADL is not the only motivation to explain its choice for modeling safety critical systems. First, the language has a precise semantics and a well defined execution model, which makes possible an automatic transformation from AADL to formal languages, such as those used in formal verification tools. Secondly, AADL runtime relies on the hypothesis taken when implementing real-time systems. For example, a subset of AADL can be mapped to synchronous language concepts which make AADL dynamic behavior deterministic.

The AADL Behavioral Annex is used to add specific real-time properties to each component of the dynamic design model and to define the software behavior at the thread level. The behavior annex has reached the final stage to be formally adopted by the SAE standardization committee.

2.1. The Language

AADL includes all the standard concepts found in Architecture Description Languages (ADL): components; connectors (used to describe the interface of components); and connections (used to link components). The set of components provided in AADL can be divided in three categories: software components (process, thread, thread group, subprogram, and data); 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 organisation 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

Communication, and the way it interacts with the scheduling of processes, is an important part of the AADL standard. AADL provides three types of ports — data, event and event data ports — that can be used to transmit data and control and describe the interface of a component. Ports are oriented: a port can be in input, output or input/output mode.

Data transmitted through ports is typed. Each input port is associated with a fresh variable that describes the state of the port. If a port has received nothing between two thread dispatches this variable is set to false. Each event or event data input port is also associated with a buffer that stores the data — or the number of events — sent through connected output ports. On thread dispatch, these inputs buffers are copied into the local memory of the thread. Some properties permit to customize the behavior of event and event data ports. For instance, the property `Queue_size` determines the maximum number of events or event data that can be received, while `Overflow_handling_protocol` describes the behavior of the port in case of overflow. There are two default policies for overflow, drop newest and drop oldest. The property `Dequeue_protocol` describes the way elements in the queue are accessed, one by one (`OneItem`) or all at once (`AllItems`).

The diagram in Figure 2 describes the interaction between data communication through ports and thread dispatching. Data ports have the simplest behavior. Data is sent at the end of the thread's execution, or at deadline, and is received at the next dispatch of the receiving thread. At the opposite, event and event data ports can send an event (resp. an event data) anytime during the execution of a thread. Events and event data are queued in the destinations ports. Input event and event data ports are delivered at the dispatch of the thread. Data communications between periodic threads can be declared as immediate or delayed. If the connection is delayed, data is sent at the deadline of the sending thread. If the connection is immediate, the receiving thread must wait the sending thread to complete. The received data will be available at the start of its (next) execution.

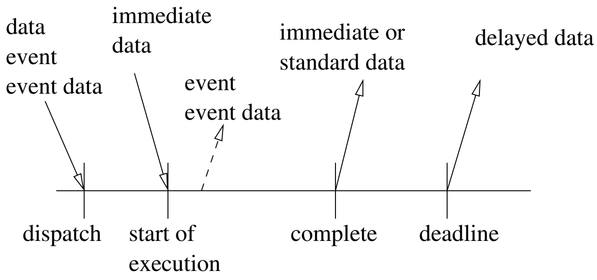

Figure 2. *communication through ports in AADL.*

2.3. Communication Through Shared Variables

As with all AADL components, data has a type and an implementation. The internal structure of the data is described in the data implementation. It is possible to specify whether different components have a shared access to a data subcomponent using the `require_data_access` connector. Correspondingly, the `provide_data_access` connector is used to state that a component allows other components access to one of its data subcomponent. The concurrency protocol used to access 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 ...

2.4. AADL Development Environments

AADL is supported by several tools like the OSATE initial framework, which has been merged into the Topcased environment, and extended with OSATE-BA, its behavior annex syntax analyser. ADELE is a graphical editor which permits to create AADL diagrams and models into Topcased, and to generate AADL source code. Beside this set of tools for the generation and lexical analysis of AADL models, we describe a methodology and a set of tools for the formal verification of AADL specifications.

3. Behavioural Verification with Tina and the AADL-FIACRE-Tina Toolchain

We briefly describe the languages and tools that support our approach for the verification of AADL models. In our approach, the AADL description is first translated in the Fiacre format, which offers a formal intermediate model to represent both the behavioral and timing aspects of the system. The Fiacre models encodes both the behavior of the AADL system as well as the AADL execution semantics. Then we compile the Fiacre model into a Time Transiton System (TTS), which is a portable, formal specification format, suitable for analysis using a model-checking tool. Therefore, Fiacre is designed both as the target language of a model transformation from AADL and as a front end to the verification toolbox.

This generic architecture as been applied in different context: transformations from different specification languages to Fiacre have already been defined — e.g. for SDL and UML — and two different verification toolboxes can be targeted, namely CADP and Tina (see Figure 1). The complete approach is based on tools that are integrated in the Topcased environment.

In the remainder of this Section we introduce the Fiacre modeling language, the Tina verification engine, and give some ideas on their capabilities.

3.1. The Fiacre Language

The design of Fiacre is inspired from decades of research on concurrency theory and real-time systems theory. For instance, its timing primitives are borrowed from Time Petri nets [6], while the integration of time constraints and priorities into the language can be traced to the BIP framework [4]. For what concerns the compositionality of the language, Fiacre incorporates a parallel composition operator and a notion of gate typing which were previously adopted in E-Lotos and Lotos-NT. We briefly describe the language. The detailed syntax and formal semantics of the Fiacre can be found in [8].

Fiacre is a strongly typed language, meaning that type annotations are exploited in order to guarantee the absence of unchecked run-time type errors. Fiacre programs are stratified in two main notions: *processes*, which describes the behavior of sequential components and *components*, which describes a system as a composition of processes, possibly in a hierarchical manner.

A program is a sequence of declarations. A process is defined by a set of control states and parameters, each associated with a set of *complex transitions*, which are programs specifying how parameters are updated and which transitions may fire. For example, the process declaration:

process `T`[`p` : **bool**](&`u` : **array** 5 **of** **bool**) **is** ...
expresses that `T` is a process that may interact over one port, `p`, which transmits boolean values, and that it has one parameters, `u`, which is a (reference to a) shared array. The behavior of a process is defined by "complex transitions", built from expressions and deterministic constructs available in classical programming languages (assignments, conditionals, while loops and sequential compositions), nondeterministic constructs (nondeterministic choice and assignments) and communication events on ports.

A component is defined as the parallel composition of processes and/or other components, expressed with the operator **par** ... || ... **end**. While components are the unit of composition, they are also the unit for process instantiation and for ports and shared variables creation. The syntax of components allows to restrict the access mode and visibility of shared variables and ports, to associate timing constraints with communications and to define priority between communication events. For example, the declaration **port** p : **bool in** [min,max] defines a port that can only interact min time units after it has been activated and must be used or deactivated before max time units (min and max should be float or integer constants).

3.2. Translation Principles

The transformation of AADL code into Fiacre relies on AADL properties and on the behavioral annex of AADL that has been developed and integrated to the OSATE AADL environment within Topcased. We follow a model-driven approach. Alongside a meta-model of AADL, we have developed a meta-model of the Fiacre language that is integrated in the Topcased tool-chain. Hence, the transformation from AADL to Fiacre can be obtained through model transformation.

We do not precisely describe the translation process in this paper, or detail the structure of the generated code. In a nutshell, we associate a Fiacre process to each AADL thread and map each AADL port to a communication port in Fiacre. (Since we focus on the behavior of the system and not its hardware architecture, we take a flattened view of the AADL model as a set of communicating threads.) Processes do not communicate directly. Events and data exchanges are mediated by a specific glue process, which manages communication and scheduling protocols. Timing information, such as the period of threads, are modelled using the time constraints mechanism provided by ports in Fiacre. More information on the translation can be found in [11].

The translation takes into account a substantial subset of the AADL standard and all basic properties are considered when generating a Fiacre model. More particularly, we take into account (1) AADL modes and priorities, as well as (2) access to shared variables. For the moment, while periods can change, we assume that priorities are fixed. We take into account that connections are determined by the current mode. On the other hand, there is currently no support for multiprocessor architecture in our translation from AADL to Fiacre. As a result, we do not take into account the value of the Actual_Processor_Binding property. We also do not handle preemption. This last feature will be added in a forthcoming version of the Fiacre language.

3.3. Behavioral Verification with Tina

Tina [7], the TIme Petri Net Analyzer, provides a software environment to edit and analyze Petri Nets and Time Petri Nets. It is particularly well suited to the verification of systems subject to real time constraints, such as those modeled using AADL.

Beside the usual analysis facilities of similar environments, the essential components of the Tina toolbox are state space abstraction methods and model checking tools that can be used for the behavioral verification of systems. This is in contrast with the broader notion of functional verification, in that we attempt to use formal techniques to prove that requirements are met, or that certain undesired behaviors cannot occur — like for instance deadlocks — without resorting to actual tests on the system. The approach followed here is that commonly referred to as *model-checking*, which basically consists in two abstract steps: (1) the generation of a formal model from a description of the system, followed by (2) a systematic exploration of the states space of this model. This involves exploring states and transitions in the model, relying on smart abstraction techniques to reduce the number and size of these states and therefore reducing the computing time.

The properties to be verified are often described in temporal logics, such as linear temporal logic (LTL) or computational tree logic (CTL). The result of the verification may lead to an accepting status, meaning that the model of the system satisfies the requirements, or exhibit an error. In the last case, it is often possible to extract a counterexample, which is an explanation at the level of the model (generally an execution trace), which leads to a problematic state. Such counterexamples could be stored alongside an AADL model.

The core of the Tina toolset — a command line tool called *tina* — is an exploration engine used to generate state space abstractions that are fed to dedicated model checking and transition system analyzer tools. The front-ends to the exploration engine convert models into an internal representation — the abstract Timed Transition Systems (TTS) — that is an extension of Time Petri nets handling data and priorities. The *frac* compiler, which converts Fiacre description into TTS and is part of the Topcased environment, is an example of such front-end.

State space abstractions are vital when dealing with timed systems, that have in general infinite state spaces. Tina offers several abstract state space constructions that preserve specific classes of properties like absence of deadlocks, linear time temporal properties, or bisimilarity. A variety of properties can be checked on abstract state spaces: general properties — such as reachability properties, deadlock freeness, liveness, . . . — specific properties relying on the linear structure of the concrete space state — for example linear time temporal logic properties, test equivalence, . . . — or properties relying on its branching structure – branching time temporal logic properties, bisimulation, . . .

Tina provides several back-ends to convert abstract state spaces into physical representations readable by the proprietary or external model checkers and transition system analyzers. Tina can present its results in a variety of formats, understood by model checkers like MEC, a mu-calculus formula checker, or behavior equivalence checkers like Bcg, part of the CADP toolset. Hence we can apply all these tools to the verification of systems modeled in AADL. In addition, several model-checkers are being developed specifically for Tina. The first available, *selt*, is a model-checker for an enriched version of State/Event-LTL, a linear time temporal logic supporting both state and transition properties. (The logic is rich enough to encode marking invariants.) For the properties found false, a timed counter example is computed and can be replayed by the simulator.

We briefly introduce the temporal logic formulas that can be checked with selt. Formulas `p, q, ...` of the logic are expressions built from the classical logical operators: negation (`-p`), conjunction (`p /\ q`), ... and the basic LTL modalities: `[]`, `<>`, `()` and `U`. A formula is said to be true if it holds on all computation paths. The formula `p` holds (relative to a computation path) if `p` holds now. That is at the start of the path. The meaning of the temporal modalities is described below.

`() p`	holds if `p` holds at the next step
`[] p`	holds if `p` holds all along the path
`<> p`	holds if `p` holds in a future step
`p U q`	holds if `p` holds until the first moment that `q` holds

Instead of requiring end-users to provide properties written in temporal logic, we propose a set of high-level *validation patterns* that simplify the elicitation of formal requirements. We give some example of verification patterns in Section 4.3: the absence of global deadlock; the unreachability of events in a set `e` (none of the events in `e` will occur during execution); the resettability of the events in `e` (events in `e` will repeat themself during execution). This pragmatic approach help us mitigate some of the complexity that is associated with the use of model-checking tools by novice users.

In complement to the temporal logic approach, realtime properties — like those expressed in so-called timed temporal logics — can be checked using the standard technique of observers, encoding such properties into reachability properties. The technique is applicable to a large class of realtime properties and can be used to analyze most of the "timeliness" requirements found in practice. For instance, using observers, it is possible to implement a validation pattern of the form `p leadsto q inlessthan t`, meaning that whenever the system is in a state where `p` holds then, before `t` units of time, property `q` will hold.

4. Experimentation

In this section, we report on our experiments carried out on the dynamic architecture for a network protocol (NPL) in charge of data communications between an airplane and ground stations. We describe the architecture of our example, the properties that have been checked and give some quantitative information. For this demonstrator, proposed by Airbus, AADL has been used to model the dynamic architecture of the NPL software subset. We show the three layers of the NPL stack in Figure 3, with a MiddleWare Protocol (MWP) mediating the communication between several high-level APplications Protocols (APP) and the underlying transfer protocol (TFTP in this case).

The NPL system includes several functions allowing the pilot and ground stations to receive and send information relative to the plane: weather, speed, destination, ... NPL, in the avionics side, is running on an IMA computer and consists of one ARINC 653 partition [3]. The NPL communicates with several other embedded computers through an AFDX field bus.

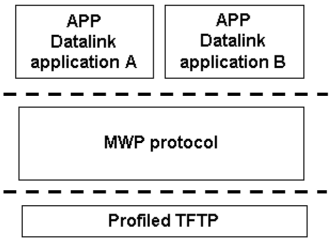

Figure 3. The Network Protocol Stack

4.1. NPL Protocol Description

The MiddleWare Protocol layer (MWP) of our system is in charge of handling messages exchanged between applications connected to the upper data-link and lower ground systems. Messages are exchanged on top of the Trivial File Transfer Protocol (TFTP). Hence, the MWP can be considered as an upper layer of TFTP and shall provide file transfer services such as: Read, Write, Abort. The list of services and requests that can be addressed to the MWP is quite rich and includes, among others, Registration, Disregistration, Downlink, Confirm, Indication, ...

The overall MWP behavior can be modelled by a communicating automaton with three main states (`closed`, `opening` and `open`) that correspond to the states of the "virtual communication" channel between the aircraft and the ground. While the number of states is small, the dynamics of the system is quite complex as it requires about sixty transitions: inputs and outputs actions of the automaton correspond to requests received or sent from/to the on-board applications or the lower ground layers. The complete NPL system is composed of several applications, and every data-link application has its own instance of the MWP automaton

The behavior of the MWP was originally defined by means of sequence diagrams describing usage scenarios in nominal and default cases. These sequence diagrams have all been checked against our automata-based specification in order to assert the correctness of our modeling. A typical usage scenario is given in Figure 4 that details a registration sequence between an application protocol (APP); the MiddleWare Protocol (MWP); the transfer protocol; and ground layers tasks (the dashed line). This is the most significant activity in the MWP since every application has to register before starting any data exchanges with ground stations. The scenarios illustrates two modes of the MWP. If the MWP is in state `closed` and receives a `registration_request` message from the APP, it initiates a connection (the MWP goes into state `opening`). If the MWP is in the `opening` state and receives a `data_indication` message from TFTP then the connection is established (the MWP enters

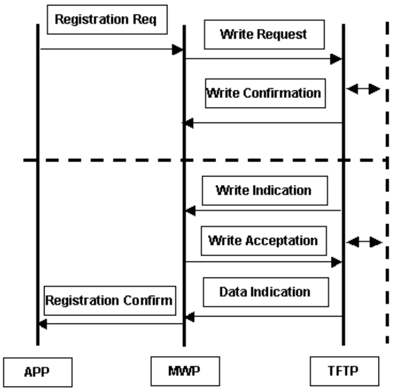

Figure 4. APP registration sequence diagram

state open).

Links between requests and states are explained below. Several sequence diagrams has been checked, in nominal or default cases, and a typical one is a registration sequence between an application, MWP, TFTP and ground layers (tasks) detailed hereafter :

4.2. Protocol Modelling with AADL

Our goal in this experiment is to model a part of a realistic system that should be executed as a single application into an ARINC 653 partition. The NPL software subset has been modeled as a single application composed of one main AADL component. The AADL code specifies both the hardware and software architecture of the component and is composed of:

- an AADL processor with its memory (AADL hardware component types);
- one main AADL process that encloses five AADL threads (AADL software component types). The allocation of software components on hardware components is defined by the AADL binding mechanism.

Software Architecture: The diagram in Figure 5 details the architecture of our system using the AADL graphical syntax. This diagram has been edited with the ADELE graphical modeler [1]. We have highlighted the five threads of the NPL component, which carry out the main functions of the application. A first thread takes care of the data-link applications (thApplis) while there is another thread for the message scheduler (thSeqMsgMWP). The remaining threads are used for: implementing the MWP state automaton (thMWP); supporting the Timer functions (thTIMER); and supporting the underlying TFTP protocol (thTFTP).

We can define the real time properties of threads by setting specific properties in the AADL specification, like for instance the dispatch protocol (periodic or sporadic), the period (time) and the deadline (time). An example of declaration can be seen in the AADL code snippet for the thread thApplis given in Listing 1. In our experiment, all the threads are periodic with periods ranging from 5 ms to 20 ms.

Communication Architecture: Communication between threads is based on mailboxes, implemented by AADL shared data access. We choose to model communications between tasks using shared data access instead of using event data ports, which are also available in AADL. While this choice complicates the description of the system, it is closer to the actual design found on an aircraft. To this end, the model includes ten shared memory buffers that are used to store the data exchanged by thread over asynchronous communication channels. These buffers are connected to the threads through external data access features. Data ports are represented by triangles while connections, represented by the lines, establish a link between a thread and a buffer.

The architecture of data connections between threads is regular. Each pair of threads, excluding the timer thTIMER), is connected through at least two buffers: one for wake-up, and another for carrying the message part. For instance, the buffer Wkup_Appli is used to store the "wake-up" signal from the MWP controller, thMWP, to the data-link applications. In our AADL specification, every memory data is set as a 32 bits integer data type.

All the threads adhere to a common communication protocol. When a thread needs to communicate with another thread, it first put its message into the dedicated buffer (for instance Prim_Appli) and then put its identifier into the associated wake-up buffer. When a thread receives an identifier into its wake-up buffer (pooling), it reads the message and then clear the identifier.

Some threads have also access to data generated outside the MWP, like for example message frames exchanged with the environment, or are connected through specific event ports (for instance, the timer and the MWP controller threads). Data exchanged with the environment are defined as structured, composite data formed from several integer fields.

Modeling Thread Behavior: The behavior of each thread can be expressed using the AADL Behavioral Annex syntax. For instance, we outline the definition for the thread thApplis in Listing 1. This code has been generated from the diagrams of Figure 5 using the ADELE code generator, except for the behavior declaration (the code in the block ANNEX behavior_specification {** ... **}) which was written manually.

The thread thApplis has five internal states : start, pending, confirm, disreg and register. The thread initial state, start, is activated only once after thread initialisation. From this state, the guard moves to register without guard conditions (but changes the value of an internal DATA component as a side-effect). From then, the thread remains in the pending state until it receives a new request (register, disreg, or confirm).

Likewise, we can use the Behavioral Annex to directly

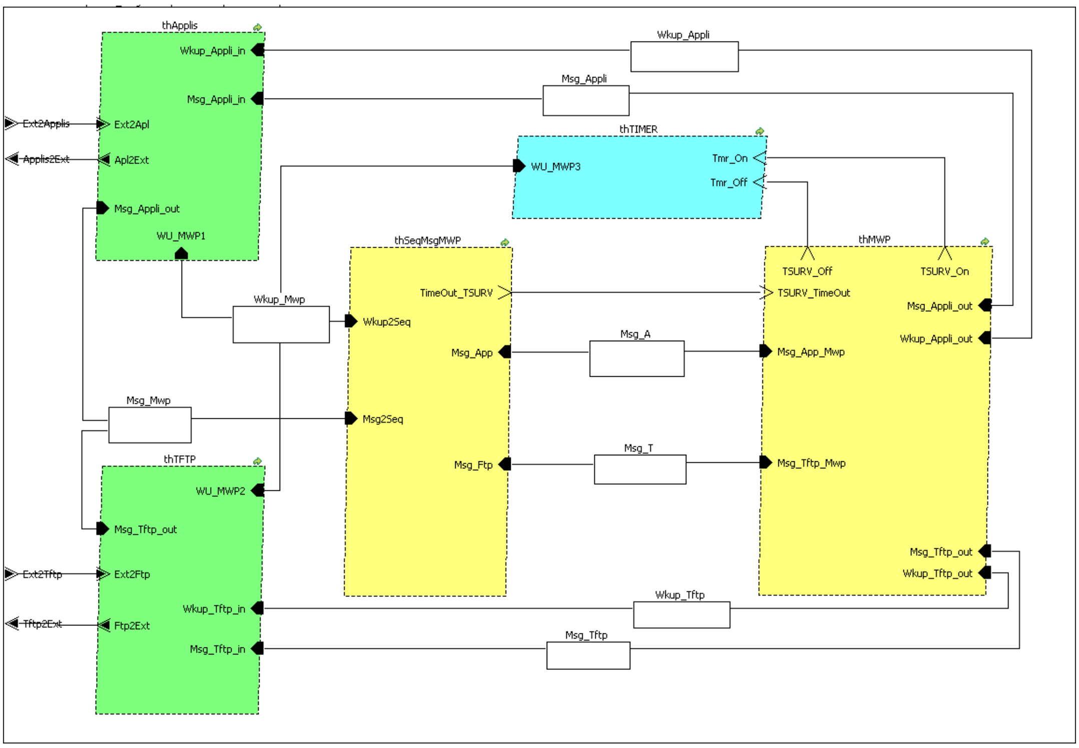

Figure 5. Graphical representation of the NPL component

```
THREAD thApplis
FEATURES
{...}
END thApplis;

THREAD IMPLEMENTATION thApplis.others
SUBCOMPONENTS
  applis2ext_msg : DATA types::msg.impl;
{...}
PROPERTIES
  Dispatch_Protocol => Periodic;
  Deadline => 10 ms;
  Period => 10 ms;
{...}
ANNEX behavior_specification {**
  states
   -- States Declaration
   start : initial state;
   pending : complete state;
   confirm : complete state;
   disreg : complete state;
   register : complete state;
 transitions
   start -[]-> pending {
     applis2ext_msg.req := 0;
     applis2ext_msg.dat := 0;
     };
   pending -[ on applis2ext_msg.req=A_Reg_Req ]->
     register { applis2ext_msg.req := 0; };
   pending -[ on applis2ext_msg.req = A_Disreg_Ind ]->
     disreg { applis2ext_msg.req := 0; };
   pending -[ on applis2ext_msg.req = A_Confirm_Ind ]->
     confirm { applis2ext_msg.req := 0; };
{...}
**};
END thApplis.others;
```

Listing 1. Example of AADL behavior description

encode the behavior of the MWP controller (thread `thMWP`) that was informally described in Section 4.1. We obtain a specification with three states (`closed`, `opening` and `open`), as given in the NPL description, and we can also specify more precisely the data that should be checked or modified within each transitions.

We do not describe more explicitly the behavior of the other threads in this extended abstract. The complete AADL specification of the MWP system requires eight graphical diagrams (of the same complexity than the one given in Figure 5). In its textual format, this amounts to about 800 lines of AADL source code with more than half of this code automatically generated from the graphical specification. On these 800 lines, the behavior of the MWP controller amounts to about 300 lines of code.

This specification can be easily reused. Hence, several applications and MWP threads could be modeled by using several instances of the same AADL specifications with update connections between them.

4.3. Functional Verification by Model-Checking

We have used the verification toolchain described in Section 3 to check properties on the AADL specification of the MWP system. The tools used to connect the AADL specification with our model-checking tools are all integrated within the Topcased platform [9], built on top of Eclipse. The Topcased environment includes parts of the formal verification toolchain through the AADL2Fiacre plug-in, which implement the transformation from AADL models into Fiacre models.

The Fiacre model obtained after transformation takes into account the complete behavior described in the AADL model. It also includes the whole AADL language execution model which means that, among other aspects, our interpretation takes fully into account the scheduling semantics as specified in the AADL standard. To achieve this goal, the behavior of each translated thread is extended with special states — `dispatch`, `schedule`, `compute` and `complete` — to represent its interaction with the scheduler. For instance, the state `schedule` corresponds to the selection of the thread by the scheduler.

After this first encoding step, the Fiacre model is compiled into a format suitable for the Tina verification toolbox that constructs an abstract state space of the complete system (the AADL components extended with the AADL execution model). The abstract state space can be explored and checked for properties expressed using temporal logic formulas. The Tina toolbox includes a model-checker for a variant of LTL (selt) and for the μ-calculus (muse).

We give more details on the properties that have been formally checked on the model. The goal was to defined a set of simple "property patterns" for dynamic architecture verification and to give them to avionics software engineers with no previous knowledge of model-checking or temporal logic. We defined three patterns that were used by system engineers to detect real-time pathologies. These properties, described bellow, have been automatically checked on the MWP model.

- `NoGlobalDeadlock`, applies to the whole model. This pattern checks for absence of global deadlocks, that is, the system can not lock himself due to a wrong synchronisation;
- `Unreachable (exp)`, applies to an internal state. This pattern checks for the presence of dead states. This is useful to check wether a thread may reach a given behavior state;
- `Resettable (exp)`, applies to a thread dispatch state. This pattern checks for healthiness, that is the fact that a given thread can be dispatched infinitely often.

These three patterns can be directly encoded in terms of the LTL-dialect used by the selt model-checker. The pattern `NoGlobaldeadlock` is expressed by the formula `[] - dead`, meaning that for every reachable state (always) it is false that no transitions can be taken from this state. The pattern `Unreachable (exp)` is equivalent to the formula `[] - (exp)`, meaning that always, the property `exp` is false. For example, the pattern `Unreachable (thApplis_pending)` can be used to test whether the thread can reach its state `pending`. Finally, the pattern `Resettable (exp)` is equivalent to the formula `[] <> (exp)`, meaning that always, we will eventually (after a finite number of transitions) enter in a state where the property `exp` is satisfied. For example, the pattern `Resettable (thApplis_dispatch)` can be used to test whether the thread `thApplis` will (always) eventually be dispatched.

In addition to these simple patterns, we have also used the full expressivity of our approach to check temporal constraints of the form: whenever the thread `A` is in state `s1` then the thread `B` will reach the state `s2` within a delay of at most `t` ms. Several scenarios have been tested with this property, for example to find an upper limit on the time needed for the completion of the sequence diagram given in Figure 4.

4.4. Performance Evaluation and Experimental Results

Experiments based on the verification toolchain were successful on the NPL use case as it was possible to verify a substantial architecture model extracted from the system described in Section 4. This model includes several threads that exchange information through shared memory data. The use of formal verification techniques at the model-level was particularly interesting in this case. Indeed, the design used in the definition of the communication architecture is prone to concurrency access problems since all threads must agree on the same order when accessing data.

With respect to performances, our verification toolchain is able to handle the generation of the complete state space of the demonstrator — which amounts to about a million of states for the Fiacre intermediate model — without any memory overflow on a typical basic development computer (Intel dual-core processor at 2 GHz clock frequency, and 2 Go of RAM memory). The abstract state space construction and system compiling are performed, on the same computer, in less than 5 minutes with a memory footprint in the order of 500 Mo of RAM. On examples of this size, the model checker included in Tina is able to prove formal properties in a few seconds. For example, it takes less than 2 minutes to check the 22 properties derived from the patterns defined in Section 4.3: one test for `NoGlobalDeadlock`; 5 resettable property (one for each thread in the system); and 16 reachability test (one for each state of each thread).

This experimentation, while still modest in size when compared to a full-blown avionic protocol, gives a good appraisal of the use of formal verification techniques for real industrial software. These experimental results are very encouraging. In particular, we can realistically envisage that system engineers could evaluate different design choices for the MWP protocol stack in a very short time cycle and test the safety of their solutions at each iteration.

5. Future Work

We have described work undertaken within the ITEA2 project Spices. This project, initiated in 2006, gathers industrial and academic partners from Belgium, France and Spain with the goal to extend and improve the usability of AADL in aeronautics, telecommunication and space domains.

During this project, several tools have been integrated to provide a complete behavioral verification chain: ADELE, a graphical AADL modeler based on Topcased framework and developed by Ellidiss Technologies; OSATE, an AADL handling back-end based on Eclipse and developed by the Software Engineering Institute; AADL2Fiacre, an Eclipse plug-in translating AADL models into Fiacre and developed by IRIT; frac, a Fiacre compiler producing Tina input models developed by LAAS-CNRS; Tina, a toolbox for Petri networks and timed transitions systems including behavioral space simplification and behavioral verification, also developed by LAAS-CNRS. This paper details how these tools can be combined in order to provide a behavioral verification environment for AADL models within Topcased.

While the methodology of our verification toolchain has already been described in previous works [11], this paper is the first occasion to report experimental study that were conducted on a significant avionic demonstrator. This experimental study is also interesting by the fact that we can draw from it a set of conclusions about the integration of model-checking tools in an industrial development process as well as several directions for extending our work. Further ideas for improvements include: to enhance and standardize our library of validation patterns; to improve behavioral modeling capabilities of ADELE (e.g. with a graphical representation of the behavioural annex); and to improve the integration of the transformation toolchain in Topcased, in particular with respect to better presenting the verification results to the end user.

References

[1] ADELE: a versatile system architecture graphical editor based on AADL. http://gforge.enseeiht.fr/projects/adele/

[2] SAE Aerospace. Architecture Analysis & Design Language (AADL). AS-5506, SAE International, 2004.

[3] Airlines electronic engineering committee. Avionics Application Software Standard Interface, ARINC Specification 653. 1997.

[4] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time systems in BIP. In *Proc. of SEFM – IEEE Software Engineering and Formal Methods*, 2006.

[5] R. B. Franca, J.-P. Bodeveix, D. Chemouil, M. Filali, D. Thomas, J.-F. Rolland. The AADL behaviour annex, experiments and roadmap. In *Proc. of ICECCS – IEEE International Conference on Engineering of Complex Computer Systems*, 2007.

[6] P. M. Merlin and D. J. Farber. Recoverability of communication protocols: Implications of a theoretical study. *IIEEE Transactions on Computers*, 24(9):1036–1043, 1976.

[7] B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research*, 42(14), 2004.

[8] B. Berthomieu, J. P. Bodeveix, M. Filali, H. Garavel, F. Lang, F. Peres, R. Saad, J. Stoecker, F. Vernadat. The syntax and semantics of Fiacre. Research Report LAAS 07264, 2007.

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

[10] The SEI AADL Team. *An Extensible Open Source AADL Tool Environment (OSATE).* Software Engineering Institute, 2006.

[11] B. Berthomieu, J.-P. Bodeveix, C. Chaudet, S. Dal Zilio, M. Filali, F. Vernadat. Formal Verification of AADL Specifications in the Topcased Environment. In *Proc. of Ada Europe 2009 – 14th Ada-Europe International Conference*, LNCS Vol. 5570, 2009.