From AADL model to LNT specification

Hana Mkaouar[1], Bechir Zalila[1], Jérôme Hugues[2], and Mohamed Jmaiel[1,3]

[1] ReDCAD Laboratory, University of Sfax, National School of Engineers of Sfax, BP 1173, 3038 Sfax, Tunisia
hana.mkaouar@redcad.org,
{bechir.zalila,mohamed.jmaiel}@enis.rnu.tn,
[2] Université de Toulouse, Institut Supérieur de l'Aéronautique et de l'Espace, BP 54032, 31055 Toulouse CEDEX 4, France
jerome.hugues@isae.fr
[3] Research Center for Computer Science, Multimedia and Digital Data Processing of Sfax, BP 275, Sakiet Ezzit, 3021 Sfax, Tunisia

Abstract. The verification of distributed real-time systems designed by architectural languages such as AADL (Architecture Analysis and Design Language) is a research challenge. These systems are often used in safety-critical domains where one mistake can result in physical damages and even life loss. In such domains, formal methods are a suitable solution for rigorous analysis. This paper studies the formal verification of distributed real-time systems modelled with AADL. We transform AADL model to another specification formalism enabling the verification. We choose LNT language which is an input to CADP toolbox for formal analysis. Then, we illustrate our approach with the "Flight Control System" case study.

Keywords: AADL, LNT, Distributed real-time systems, Architecture description languages, Model transformation, Specification languages, Formal verification

1 Introduction

Building distributed real-time systems is a tedious task. They are usually complex systems, often used in safety-critical domains like avionics and aerospace. Such systems must satisfy both real-time constraints and other constraints imposed by the distribution of nodes. Several solutions have been introduced to simplify the development process through modeling and code generation thanks to Architecture Description Languages (ADLs). These languages allow the description of structure, behaviour and configuration offering an abstract view of the entire system. The development process of distributed real-time systems requires verification in earlier phases to ensure the correctness of the produced system. For this purpose, designers have joined verification and validation formalisms with ADLs which are considered like a pivot language.

Formal methods are widely used to check rigorously a critical system. They are used to confirm if the system satisfies the user needs (validation) and if it

complies with its specification (verification). However, a system modelled with an ADL cannot be directly formally verified. The use of such methods requires a formal specification of the checked system. Several formalisms are considered in the formal world for example Petri net, automata and process algebra. In this context, researches are directed towards the transformation of the architectural models into other models in order to connect with formal verification tools.

AADL [3] (Architecture Analysis and Design Language) is a rich and complete ADL for embedded real-time systems, with an emphasis on critical avionics systems. Many work in the literature apply the transformation model alternative for AADL formal verification: they transform various AADL subsets into different specification formalisms and they focus on behavioral analysis by checking general properties like deadlock with model-checking.

In our proposed approach, we include formal verification in development process of systems modelled with AADL. Our work is integrated in the Ocarina [11] tool set which is a development environment for AADL modeling and code generation. We adapt process algebra formalism for the transformation which concerns an interesting AADL subset for communication and scheduling semantics.

We choose LNT [5] as target specification language which derives from two standards Lotos [1] and E-Lotos [2]. This choice is justified by the expressiveness and richness of LNT. It provides expressive enough operators for data and behaviour description and it has a user-friendly notations to simplify the specification writing. Indeed, LNT is a CADP [6] (Construction and Analysis of Distributed Processes) input language. It is a popular formal verification toolbox that implements many formal methods.

In this paper, we report our proposed transformation AADL/LNT and we prove its effectiveness with the "Flight Control System" case study. The remainder of this paper is organized as follows: Section 2 gives an overview of AADL then presents LNT language. In section 3, we detail the translation of AADL model into LNT specification. In section 4, we present tools used in our work. Section 5 applies transformation rules on our case study. In section 6, we discuss related work. Finally, conclusions and future work end the article in section 7.

2 Preliminaries

In this section, we briefly introduce AADL with the considered subset. Then, we present our target specification language LNT.

2.1 AADL

AADL [3] is an industrial ADL for critical domains like avionics, aerospace and automotive. It is standardized by the SAE (Society of Automotive Engineers), the last version (version 2) was published in 2009. AADL is a rich language with a textual syntax and graphical representation. It allows the modeling of the structure, behaviour and configuration of distributed real-time embedded

systems. Like most of ADLs, AADL consists of three basic elements: components (software, hardware and system), connections (to link components) and the description of the architecture configuration with AADL properties.

Components. An AADL component is defined through a type (it declares the component interface elements called features) and zero or more implementations (they present the component internal structure). AADL defines three categories of components: software components (data, thread, thread group, subprogram, subprogram group and process), hardware components (memory, bus, virtual bus, processor, virtual processor and device) and system component. We briefly describe the subset of AADL components considered in our work.

Software components present the applicative part of the system. In our approach, we consider these components: `data` represents a data type within the system; `subprogram` represents sequentially executed source text which can be coded in programming languages like C and Ada language; `thread` is a concurrent schedulability unit of sequential execution through source code. A thread always executes within a process; `process` represents a virtual address space which contains thread and data associated with the process and with its subcomponents.

Hardware components present the computing hardware and the physical environment. In our work, we consider the following components: `bus` represents hardware and associated communication protocols to exchange control/data among other execution platform components; `processor` is an abstraction of hardware and software for scheduling and execution of threads.

System component represents a composite of software and hardware components or system components.

Connections. AADL connection is a linkage established between component features to exchange data and control. There are four categories of features: port, subprogram, parameters, and subcomponent access. They enable three types of connections: port connection, parameter connection and access connection. In our work, we are interested in the port connection type.

Port connection presents the transfer of data and/or control between two components, explicitly declared between two ports. There are three types of ports in AADL: data, event and event data. Ports are typed with a data component (the type of transferred data) and they are directional.

Properties. AADL properties provide additional information about AADL elements (component types/implementations, connections, etc). We distinguish: properties specifying constraints for hardware binding for example `Actual_Proce ssor_Binding` property to bind thread with the processor and `Actual_Connectio`

`n_Binding` to bind connections to the bus; properties specifying temporal information like `Period` and `Dispatch_Protocol` for threads and `Scheduling_Protocol` for processor; and properties specifying information for ports such as `Dequeue_Protocol`, `Input_Time` and `Output_Time` to model event processing policies and their time of arrival.

2.2 LNT

LNT [5] is a specification language for safety-critical systems developed by the Vasy team in INRIA. The latest version (Version 6.1) was published in 2014. It is a heir of Lotos [1] language and a simplified variant of E-Lotos [2]. It combines strong theoretical foundations of process algebra with features from imperative and functional programming languages. LNT is supported by the CADP toolbox which offers a rich formal verification like simulation and model checking.

LNT is similar to CSP and CCS process algebra, it represents a system by a set of processes communicating through channels. An LNT specification consists of two parts: the data part defines types and functions; and the control part defines the behaviour (process). The control part is a super-set of data part. It includes all data part instructions and adds the non-determinism, the asynchronous parallelism and communications. In the rest of this section, we present the essential elements of the LNT language. We include some definitions to make the comprehension easier. In the rest of this paper, we adopt the following notations: B for behaviour; G for gate identifier and P for process identifier.

Module. In LNT, the system is modelled by a set of parallel process in communication through communication ports. It is defined in a module, with the same name as the file in which it is contained. It can import others modules. A process called "MAIN" defines the entry point of the specification.

Process. An LNT process, whose the definition is included in listing 1.1, is an object that describe a behaviour, it can be parameterized by a list of formal gates, a list of formal variables and a list of formal exceptions. LNT allows to describe several behaviours such as sequential composition, non-deterministic assignment, conditional behaviour, non-deterministic choice and parallel composition.

Listing 1.1. LNT process definition

```
process_definition ::= process P
   [[gate_declaration_0,.., gate_declaration_m ]]
   [(formal_parameters_0,.., formal_parameters_n )]
   [raises exception_declaration_0,.., exception_declaration_k]
is
   B
end process
```

Parallel composition. LNT processes can be combined in parallel and synchronized on gates with the `par` instruction. Parallel processes start execution

and terminate in the same time without preemption. `par` instruction, given in the listing 1.2, allows two types of synchronization: the global synchronization (defined with $G_0...G_n$), this communication can happen only if all processes can make it simultaneously; and the interface synchronization, in this case, if a process is waiting for a communication in a gate belongs to its synchronization interface ($G_{(i,0)}, ..., G_{(i,n_i)}$), this communication can happen only if all process that contain this gate in their synchronization interface can make this communication simultaneously.

Listing 1.2. LNT parallel composition

```
par [G_0 , ... , G_0 in]
    [G_(0,0) , ... , G_(0,n_0) -> ] B_0
    || ... ||
    [G_(m,0) , ... , G_(m,n_m) -> ] B_m
end par
```

Communication. In LNT, processes communicate by rendezvous on gates. LNT gate can be typed with channel. A channel defines a set of gate profiles. With the same gate, process can send and receive messages. The communication is blocked in sending or receiving. In effect, the process waiting for a communication is suspended and terminates after the rendezvous takes place.

3 Transformation Rules

Model transformation plays an essential role in model-driven engineering for various purposes such as modeling, optimization and analysis. It defines a mechanism for generating a target model based on information extracted from a source model. One important issue in this domain is the semantic preservation that should be considered while defining the transformation description. In our approach, we provide a model transformation description with a set of model transformation rules from AADL to LNT for formal verification. We abstract AADL model as a set of communicating execution units in real-time context. Precisely, we consider port communication between AADL threads enriched with scheduling properties. We find LNT language suitable and expressive enough to specify AADL semantic in different aspects such as hierarchy of components, parallel execution and connection types.

Basically, we translate every AADL component (type/implementation) into an LNT process. We present AADL port connections with LNT gates and we implement AADL properties with LNT programming structures. With this strategy, we extract the three ADL basic elements (seen in section 2.1) from AADL model. So we obtain a specification with the same structure of the initial model.

3.1 Scheduling Mapping

In our approach, we consider periodic thread scheduling and execution mechanism without sharing resources. Our mapping considers the schedulability test

like a primary condition before checking other constraints. It concerns AADL thread and processor components detailed in below sections.

Periodic Thread. First rule (table 1) in our transformation description concerns AADL thread. Every implementation of thread component is presented with an LNT process `<Thread_*>`. It has a set of gate declarations corresponding to AADL ports with a set of parameters corresponding to AADL temporal properties like execution time, input/output time, period and priority.

Table 1. Transformation rule for AADL Thread

Rule 1	

Thread
- Features
• data/event/event data port
- Properties
• `Period`;
• `Compute_Execution_Time`;
• `Deadline`;
• `Input_Time`;
• `Output_Time`;
• `Priority`.
	`process <Thread_AADLIdentifier>`
	`[OutR : Request,`
	`InR : Response,`
	`-- AADL gate declarations`
	`]`
	`(LCM : Nat,`
	`priority : Nat,`
	`Destination : Connections,`
	`Idt : Identifier,`
	`Execution_Time : Time,`
	`Input_Time : IO_Time_Spec,`
	`Output_Time : IO_Time_Spec,`
	`Period : Nat,`
	`Deadline : Nat)`

Initially, all `<Thread_*>` processes are considered in the ready state. To start the execution and enter the running state, every `<Thread_*>` contacts the processor, it requests time corresponding to its `Activate_Execution_Time` parameter. Depending on processor scheduling, `<Thread_*>` can be in three behaviours:

- Starting execution and remaining in the running state until the completion of execution in the current dispatch;
- In the case of a completion, `<Thread_*>` enters the awaiting dispatch state for the next dispatch;
- In the case of a preemption, `<Thread_*>` returns to the ready state to request the needed time to complete execution;
- In the case of exceeding its deadline, `<Thread_*>` declares (with specific gate) the failure of schedulability test and stop its execution.

LNT language is not a specific process algebra for real-time systems, it has no time operators and no preemptions. So we use counters to present time. We perform calculation, based on AADL property values, to deduct needed values like dispatching and communication times.

Processor. For Rule 2 (table 2), we extract the scheduler of AADL processor providing thread scheduling functionality. It becomes an LNT process `<Processor_*>` with two gates: the first one for receiving `<Thread_*>` requests and the second for sending the response. This process complete the scheduling mapping by computing start and completion execution time for each bounded thread.

Table 2. Transformation rule for AADL Processor

Rule 2	

Processor
 	
- Features
• requires bus access
- Properties
• `Scheduling_Protocol`
	`process <Processor_AADLIdentifier>`
	`[Input : Request,`
	`Output : Response_List] is`
	`-- code`
	`end process`

When starting, `<Processor_*>` receives requests from all ready threads which are queued and sorted by priority. After all calculations, it sends a response to enable the execution. Thus, each bounded thread gets its execution time and then starts the running state.

Thanks to its programming ability, we can implement many scheduling algorithms with LNT. We developed the Rate-monotonic scheduler for periodic threads: threads can be preempted; they share no resources; their deadlines are equal to their periods; and they have static priorities.

3.2 Communication

Port feature and port connection semantic are well detailed in AADL standard. We are interested in port communication between threads and processes. Port declarations are transformed in gate declarations in LNT processes. However, port connections cannot be transformed directly in gate synchronizations. LNT provides a rendezvous communication that cannot present AADL semantic port connection in which inputs and outputs are frozen. For example, incoming data, event or event data are not available to the receiving thread until the next dispatch (the default input time). So, we do not synchronize `<Thread_*>`s directly on gates. In addition, LNT language does not provide queues with its gates. To obtain the closest behaviour, we add an auxiliary LNT process (table 3) to present connections and handle queues. Thus, `<Thread_*>` is never blocked in a communication and ports can stack inputs in the case of exchanging events. The additional generic process `<*Port_*>` has two gates: the first one for inputs (can be from Bus, Thread or Process) and the second one for outputs (can be data or a list of event/event data). `<*Port_*>` implements connection properties: type

of communication: `data`, `event`, `event data`; queue size; overflow handling protocol: drop oldest, newest drop, error; queue protocol: FIFO, LIFO and dequeue protocol: one item, multiple items, all items. Particularly, `<DataPort_*>` presents sampled data port connection, which is a specific port connection semantic for data ports and periodic threads.

Table 3. Transformation rule for AADL communication

Rule 3	

Port
 	
- Types
• data/event/event data
- Properties
• `Queue_Processing_Protocol`;
• `Queue_Size`;
• `Overflow_Handling_Protocol`;
• `Dequeue_Protocol`;
• `Dequeued_Items`.
 	
Port connections
 	
- Port Connection Topology
• n-to-n for event/event data port
• 1-to-n for data port
- Sampled Data Port Communication
• Immediate
• Delayed
	`process <EventPort_AADLIdentifier>`
	`[Input : ChannelI,`
	`Output : ChannelII]`
	`(ConnectIDs : Connections,`
	`Queue_Size : Nat,`
	`Overflow_Handling_Protocol :`
	`Overflow_Handling_Protocol_Type,`
	`Queue_Processing_Protocol :`
	`Queue_Processing_Protocol_Type,`
	`Dequeue_Protocol :`
	`Dequeue_Protocol_Type,`
	`Dequeued_Items : Nat)`
	
	`process <DataPort_AADLIdentifier>`
	`[Input : ChannelI,`
	`Output : ChannelII]`

In LNT, gates are bidirectional. So we can present all AADL port directions: in, out and in out. The correspondence between in/out ports is ensured with identifiers, every connection (from out to in port) has an identifier. Sender includes a list of connection identifiers in its output. `<EventPort_*>` has a list of accepted connection identifiers (parameter `ConnectIDs`) to verify if its `<Thread_*>` is concerned by the received input. Thus, we can specify all AADL connection topologies. The Example presented in listings 1.3 and 1.4 transforms Producer/Consumer communication: `Producer` provides inputs to two consumers `Consumer1` and `Consumer2`. This is an 1-to-n topology and event data type AADL port connection. In LNT, we get five processes. `Process_Producer` sends messages. Each `EventPort_ConnPC*` identifies its concerned inputs with identifiers `Producer_D__Consumer*_D`.

Listing 1.3. AADL initial model

```
system implementation S.Impl

subcomponents
   Producer : process A.Impl;
   Consumer1 : process B.Impl;
   Consumer2 : process B.Impl;
   -- code
connections

   -- code
   ConnPC1 : port
      Producer.D -> Consumer1.D;
   ConnPC2 : port
      Producer.D -> Consumer2.D;
   -- code
end S.Impl;
```

Listing 1.4. LNT obtained model

```
par
   Process_Producer [..] (
        -- parameters
     {Producer_D__Consumer1_D ,
      Producer_D__Consumer2_D })
   ||
   -- code
   EventPort_ConnPC1 [..] (
        -- parameters
     {Producer_D__Consumer1_D })
   ||
   -- code
   EventPort_ConnPC2 [..] (
        -- parameters
     {Producer_D__Consumer2_D })
end par
```

3.3 Parallel composition

Process and System AADL components are organized into a hierarchy of subcomponents: process may contain a composition of threads and system presents a composition of components. To preserve this structure, we translate these hierarchical organizations using `par` behaviour with rule 4 (table 4).

Table 4. Transformation rule for parallel composition

Rule 4	

Process
 	
- Features
  • data/event/event data port
- Subcomponents
  • Thread
- Connections
  • Port connection
	`process <Process_AADLIdentifier> [`
	`  -- AADL gate declarations`
	`] is`
	`  par .. in`
	`  <Thread_AADLIdentifier> [..]`
	`  ||`
	`  <Thread_AADLIdentifier> [..]`
	`  || ..`
	`  end par`
	`end process`

System
 	
- Subcomponents
  • Process
  • Processor
  • Bus
- Connections
  • Port connection
  • Bus access
	`process Main is`
	`  par .. in`
	`  <Process_AADLIdentifier> [..]`
	`  ||`
	`  <Bus_AADLIdentifier> [..]`
	`  ||`
	`  <Processor_AADLIdentifier> [..]`
	`  || ..`
	`  end par`
	`end process`

For AADL process that contains a composition of threads. It becomes an LNT process containing a composition of <Thread_*>s. Else, there is no need to transform AADL process. Similarly, we apply rule 4 on AADL system component. It becomes the LNT "Main" process which composes process instances of transformed software and hardware subcomponents.

3.4 Synchronization

All transformed components should be synchronized to assemble the whole system. For synchronization, we define the following rules:

Rule 5 For each in/out communication, we apply two level of synchronization:

I/ every in port of <Thread_*> is synchronized with an instance of <*Port_*> process;

II/ the obtained composition from I/ is synchronized with <Thread_*> of out port.

In listing 1.5 and 1.6, we give an example where we apply rule 4 for two communicating AADL threads ThreadA and ThreadB. We obtain three synchronized processes: a first "par" composition (Thread_threadB in global synchronization with DataPort_ConnAB on gate SyncI) is in synchronization with Thread_threadA (on gate SyncII).

Listing 1.5. AADL initial model

```
thread ThreadA
features
   D : out data port DataAB;
end ThreadA;
thread ThreadB
features
   D : in data port DataAB;
end ThreadB;

process implementation P.Impl
subcomponents
   threadA : thread ThreadA;
   threadB : thread ThreadB;
connections
   ConnAB : data port
   threadA.D -> threadB.D;
end P.Impl;
```

Listing 1.6. LNT obtained model

```
process Process_P_Impl [
  -- gate declarations
  ]
is
-- code
par SyncII in
   Thread_threadA [SyncII]
   ||
   par SyncI in
      Thread_threadB [SyncI]
      ||
      DataPort_ConnAB
        [SyncII, SyncI]
   end par
end par
--  code
end process
```

Rule 6 (table 5) Gates are synchronized in competition access or simultaneous access: all <Thread_*> are in competition access to <Bus_*> for sending messages (respectively to <Processor_*> for sending requests) and in simultaneous access for receiving messages (respectively for receiving responses).

3.5 Other transformation rules

In this section, we complete the presentation of our transformation description. It concerns Data, Subprogram and Bus AADL components. Due to the lack of space, we expose briefly the rest of rules.

Table 5. Bus and Processor binding transformation

Rule 6	

Bus access
 	
- Properties
• `Actual_Connection_Binding`;
 	
Processor binding
 	
- Properties
• `Actual_Processor_Binding`;
	`par`
	`  inBus, outBus ->`
	`  <Bus_> [inBus, outBus]`
	`  ||`
	`  Rq, Rs -> <Processor_> [Rq, Rs]`
	`  ||`
	`  Rq, Rs, inBus, outBus ->`
	`  par Rs in`
	`  par`
	`  <Thread_> [Rq, Rs, inBus]`
	`  || ..`
	`  end par`
	`  ||`
	`  par Rs, outBus in`
	`  <Thread_> [Rq, Rs, outBus]`
	`  || ..`
	`  end par`
	`  end par`
	`end par`

Rule 7. *Data* is considered as a simple data type without features. It is translated to a suitable LNT type `<Data_*>` in order to present the exchanged data between threads. In addition, we add corresponding channels for gate communications.

Rule 8. *Subprogram* becomes an LNT process without gates, containing the same parameters as AADL subprogram. We can translate code given in the programming language of the source text (Ada or C). Thus, subprogram calls are translated into a simple process instantiation in the corresponding `<Thread_*>`.

Rule 9. *Bus* becomes an LNT process `<Bus_*>`, with two ports (input, output) modeling a queue with a capacity determined by a parameter. `<Bus_*>` uses the push type where the communication is initiated by the sender. It allows the bound of any connection category: data/event/event data connection and immediate/delayed connection. `<Bus_*>` exchanges a message contained the following information: sender identifier, list of connection identifiers, exchanged data and data sending interval time.

4 Tools

Our contribution benefits from a couple of powerful tools Ocarina for modeling and CADP for analyzing:

Ocarina is a tool set designed in Ada for AADL modeling. It provides syntactic and semantic analysis, verification and code generation from AADL models in Ada and C languages. Ocarina compiler has two parts a frontend and a backend. Frontend analyzes AADL model and presents it as Abstract Syntax Tree (AST). Backend treats AADL AST to produce all types of generation.

We began our implementations in Ocarina backend (implementation details will be exposed in a forthcoming paper). We do not use model transformation languages for our generation. We apply directly transformation rules on AADL AST to generate an LNT AST. Then, we scan this tree in order to produce the corresponding LNT code file.

CADP is a toolbox for the design and verification of concurrent systems. It supports several specification languages (Lotos, Fsp, LNT, etc). It includes many tools for formal analyzing and bug detection like model checking, equivalence checking, simulation, performance evaluation, etc.

We consider CADP like a black-box. Yet, we should provide all inputs: our translation generates LNT file, additional inputs must be presented depending on the concerned tool. For example, model-checker tool verifies if LNT specification satisfies a property expressed in temporal logic. In this case, we also specify a set of properties as a second input. After analyzing, CADP gives useful results for the correction of the initial model. For example, model-checker gives a false/true response for every checked property.

5 Case Study

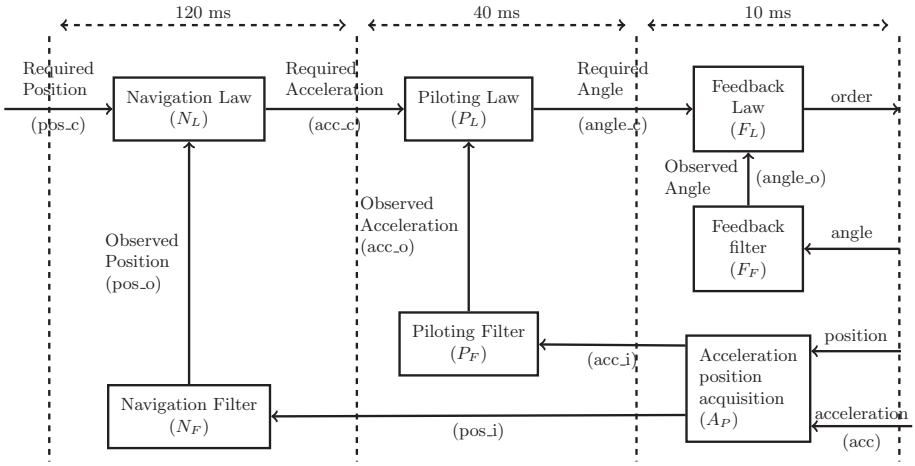
In this section, we test our contribution with the "Flight Control System" case study. We apply our transformation rules on the given AADL model to obtain an LNT specification which can be compiled and checked with CADP toolbox.

5.1 Flight control system

This system allows the control of the altitude, the speed and the trajectory of an airplane. It consists of seven periodic threads grouped in one process binding to a processor. Threads, shown in figure 1, communicate directly and exchange data. FL thread acquires the state of the system (angles, position, acceleration) and computes the feedback law of the system. The order is then sent to the flight control surfaces. PL and PF threads determine the acceleration to apply. NL and NF determine the position to reach.

Flight control system in LNT. After transformation, we obtain an LNT specification which is formed of 19 composite processes. In this example, threads communicate without bus and exchange data. So the obtained specification uses `DataPort_*` for inputs and `Processor_CPU` for RMS scheduling.

Fig. 1. Flight control system



The listing 1.7 contains an extract from the `Process_FCS` specification, showing the instantiation of `Thread_NL` and `Thread_NF` (without parameters) in synchronization with `Processor_CPU` (Rule 6). For example, `Thread_NL` has three connections (Rule 5.I): output with `acc_c` gate; input 1 through synchronization with `DataPort_Posc`; and input 2 through synchronization with `DataPort_Poso`. Also it is in synchronization with `Thread_NF` for `pos_o` input (Rule 5.II).

Listing 1.7. Flight control system in LNT

```
process Process_FCS [Rq : Response, Rs : Request,
  pos_c : Channelpos_c, order : Channelorder, acc : Channelacc,
  position : Channelposition, angle : Channelangle
  ] is
-- code
   par Rs in
      acc_c, pos_o ->
      par
         SyncI_i, SyncI_j -> Tread_NL [Rq,Rs,SyncI_i,SyncI_j,acc_c](..)
         ||
         SyncI_i -> DataPort_Posc [pos_c, SyncI_i]
         ||
         pos_o, SyncI_j -> DataPort_Poso [pos_o, SyncI_j]
      end par
      ||
      pos_o, pos_i ->
      par SyncI_k in
         NF [Rq, Rs, SyncI_k,pos_o](..)
         ||
         pos_i -> DataPortAADL [pos_i, SyncI_k]
      end par
      ||
      -- code
   end par
end process
```

5.2 Verification with CADP

The generated LNT specification can be analyzed with different tools in CADP. CADP transforms the obtained LNT specification into an LTS (Labeled transition system). In addition, CADP offers an automated reduction which allows a strong reduction in the state space. We include, in table 6, the state space statistics of the generated LTS for our case study.

Table 6. "Flight Control System" LTS

	LTS	Reduced LTS
states	569 740	59 648
transitions	4 140 014	506 791

We can simulate LNT specification with CADP simulators like OCIS (Open/-Caesar Interactive Simulator). We can check various constraints with CADP model-checkers. In our case study, we use the Evaluator 3 model-checker [12] and we express properties in Rafmc (Regular Alternation-Free Mu-Calculus) language. For example, we verify the deadlock freedom:

$[true*]\ <true>\ true$

We can check the reachability of any state, for example, the following property verifies if the order is finally sent by FL thread:

$<\ true*\ .\ "ORDER\ !FL\ "\ >\ true$

To check the schedulability of the AADL system, we add a specific LNT gate `Is_Schedulable`. If the execution ends successfully, the `<Thread_*>` writes `TRUE` in `Is_Schedulable` gate. Else, it writes `FALSE` when it detects an exceeding of deadline. Then, the model-checker verifies if this gate has the value `FALSE`. In our case study, this property is expressed in Rafmc as following:

$[true*\ .\ "IS\_SCHEDULABLE\ !Thread\_NL\ !FALSE"]\ false\ and$
$[true*\ .\ "IS\_SCHEDULABLE\ !Thread\_*\ !FALSE"]\ false\ and$
$\ldots$
$[true*\ .\ "IS\_SCHEDULABLE\ !Thread\_AP\ !FALSE"]\ false$

We model-checked the "Flight Control System" case study, and ensure that it is well scheduled and has no deadlocks.

6 Related Work

Several work in formal verification of AADL models have been made by translating AADL with or without its annex into several specification languages: (i)transformation into Petri nets for example the symmetric net in [9] for model-checking; (ii) transformation into automata for example the use of timed automata in [8] and [10] to connect to the model-checker UPPAAL and the use of

the linear hybrid automata in [7] for schedulability analysis; and (iii) transformation into different process algebras.

Our work is included in group (iii) with others approach, such as, the translation into Bip [13] to connect with Bip framework. This translation generates timed Bip models which should be transformed into non-timed models to be analyzed.

[4] transforms AADL model into Fiacre model for behavioral verification with Tina tool set or CADP toolbox. For the second alternative, AADL model is firstly converted into a Fiacre model and then transformed into a Lotos specification with the need of manual improvements. This work ignores hardware components and it is restricted on no preempted thread without scheduling execution.

[14] uses the Real-Time Maude language for transformation and the Maude framework for verification. However, this work focuses only on the software AADL components and ignores scheduling information.

Authors in [15] present a verified transformation of AADL model to TASM language using TASM Toolset, Coq and UPPAAL. This work considers only a synchronous subset of AADL (periodic threads with data port communication).

In our approach, we provide an automated model transformation of AADL models. We consider a subset of AADL language implicating software and hardware components with a significant property set. We focus on thread scheduling execution and port communication mechanism with the definition of an explicit scheduler. We use directly LNT input language of the formal tool without additional transformation. For verification, our work allows the connection with the CADP toolbox which offers a various verification methods and avoids the state explosion problem with a compositional verification.

7 Conclusion and future work

We presented our approach in the context of the verification of distributed real-time systems. We proposed a solution that allows the verification of AADL models using formal methods known by their rigorous checking results. We translated an interesting subset of AADL model to an LNT specification to exploit the CADP toolbox. Our mapping abstracts AADL model as a set of scheduled threads in communication enriched with connection and timing properties.

This paper introduced a first step of our contribution and validated its feasibility. Currently, we are focusing on communication problems. We plan to describe specific properties for communication consistency verification. Also we aim to exploit the compositional verification offered by CADP. And we are working continuously in our implementations in Ocarina.

Acknowledgments The idea of translating AADL to LNT was first explored by Hubert Garavel from the CADP group. We would like to thank him and Wendelin Serwe and Frédéric Lang for their help in using LNT and CADP.

References

1. ISO/IEC. LOTOS a formal description technique based on the temporal ordering of observational behaviour. International Standard 8807, International Organization for Standardization Information Processing Systems Open Systems Interconnection, Geneve. 1989.
2. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization Information Technology, Geneve. 2001.
3. AS5506A: Architecture Analysis and Design Language (AADL) Version 2.0. 2009.
4. B. Berthomieu, J.-P. Bodeveix, S. Dal Zilio, P. Dissaux, M. Filali, P. Gaufillet, S. Heim, and F. Vernadat. Formal Verification of AADL models with Fiacre and Tina. In *ERTSS 2010 - Embedded Real-Time Software and Systems*, pages 1–9, TOULOUSE (31000), France, May 2010. 9 pages DGE Topcased.
5. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, C. McKinty, V. Powazny, W. Serwe, and G. Smeding. Reference manual of the lnt to lotos translator. 2014.
6. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. Cadp 2011: a toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, 2013.
7. S. Gui, L. Luo, Y. Li, and L. Wang. Formal Schedulability Analysis and Simulation for AADL. In *ICESS*, pages 429–435, 2008.
8. M. E.-K. Hamdane, A. Chaoui, and M. Strecker. Toolchain Based on MDE for the Transformation of AADL Models to Timed Automata Models. 2013.
9. M. Hecht, A. Lam, and C. Vogl. A Tool Set for Integrated Software and Hardware Dependability Analysis Using the Architecture Analysis and Design Language (AADL) and Error Model Annex. In *ICECCS*, pages 361–366, 2011.
10. A. Johnsen, K. Lundqvist, P. Pettersson, and O. Jaradat. Automated Verification of AADL-Specifications Using UPPAAL. In *HASE*, pages 130–138, 2012.
11. G. Lasnier, B. Zalila, L. Pautet, and J. Hugues. Ocarina: An environment for aadl models analysis and automatic code generation for high integrity applications. In *Reliable Software Technologies–Ada-Europe 2009*, pages 237–250. Springer, 2009.
12. R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*, 46(3):255–281, 2003.
13. A. R. M. B. Mohamed Yassin Chkouri and J. Sifakis. Translating AADL into BIP - Application to the Verification of Real-Time Systems. In *MoDELS Workshops*, pages 5–19, 2008.
14. P. C. Ölveczky, A. Boronat, and J. Meseguer. Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude. In *FMOODS/FORTE*, pages 47–62, 2010.
15. Z. Yang, K. Hu, D. Ma, J.-P. Bodeveix, L. Pi, and J.-P. Talpin. From aadl to timed abstract state machines: A verified model transformation. volume 93, pages 42–68. Elsevier, 2014.