An Ocarina Extension for AADL Formal Semantics Generation

Hana Mkaouar
ReDCAD Laboratory, University of Sfax, National School of Engineers of Sfax
Sfax, Tunisia
hana.mkaouar@redcad.org

Bechir Zalila
ReDCAD Laboratory, University of Sfax, National School of Engineers of Sfax
Sfax, Tunisia
bechir.zalila@enis.tn

Jérôme Hugues
Université de Toulouse, Institut Supérieur de l'Aéronautique et de l'Espace
Toulouse CEDEX 4, France
jerome.hugues@isae.fr

Mohamed Jmaiel
Research Center for Computer Science, Multimedia and Digital Data Processing of Sfax
Sfax, Sakiet Ezzit, Tunisia
mohamed.jmaiel@enis.rnu.tn

ABSTRACT

The formal verification has become a recommended practice in safety-critical software engineering. The hand-written of the formal specification requires a formal expertise and may become complex especially with large systems. In such context, the automatic generation of the formal specification seems helpful and rewarding, particularly for reused and generic mapping such as hardware representations and real-time features. In this paper, we aim to formally verify real-time systems designed by AADL language. We propose an extension *AADL2LNT* of the Ocarina tool suite allowing the automatic generation of an LNT specification to draw a gateway for the CADP formal analysis toolbox. This work is illustrated with the *Pacemaker* case study.

KEYWORDS

Real-time systems, model transformation, formal verification, AADL.

ACM Reference Format:
Hana Mkaouar, Bechir Zalila, Jérôme Hugues, and Mohamed Jmaiel. 2018. An Ocarina Extension for AADL Formal Semantics Generation. In *Proceedings of ACM SAC Conference, Pau,France, April 9-13, 2018 (SAC'18),* 9 pages.
https://doi.org/xx.xxx/xxx_x

1 INTRODUCTION

The verification of real-time systems is a challenging topic, especially when they are used in safety-critical domains where one failure can result in serious damages. Recently, formal methods have become one of the advocated techniques in safety-critical software engineering. To be formally verified, the system should be firstly specified with a specific formalism and so it can be explored with analysis tools. However, the hand-written of the formal specification requires a formal expertise and may become complex and tedious task especially with large systems. Therefore, it is helpful to provide an automatic generation of the formal specification to assist designers in system verification. For the same purposes, systems designed with architectural languages such as AADL, MARTE and SysML are transformed into other formalisms like Petri nets, automata and process algebras, in order to draw gateways for verification tools.

In this context, we aim at the automation of the formal verification of systems modeled with AADL [1] (Architecture Analysis & Design Language). AADL is an industrial standard[1] language for embedded and real-time system modeling. Many projects and tool-chains are dedicated to AADL processing, that often adopt model transformation techniques to allow formal analyses. Several AADL formal semantics are defined using different formalisms such as Petri nets [21, 23], timed automata [12, 14] and different process algebras [4, 5, 8, 20, 25]. These approaches are implemented within platforms such as OSATE and TOPCASED aiming the convenient reuse of existing tools like UPPAAL, Tina and Polychrony. In our work, we choose Ocarina [16], a tool suite that gathers analysis of AADL models and the automatic code generation towards AADL runtimes PolyORB-HI/Ada or C. We define a model transformation *AADL2LNT* implemented within Ocarina to automatically generate a formal specification from the AADL model using the LNT [7] language. LNT[2] is a process algebra based on two standards Lotos and E-Lotos. This language provides a simplified and user-friendly syntax with expressive operators for data and behavior, suitable for the mapping of complex real-time features (tasks, scheduling algorithm, etc). Moreover, we aim to provide an automatic verification based on model-checking of a set of generic properties with the CADP[3] [11] (Construction & Analysis of Distributed Processes) toolbox. CADP is a well experimented tool for formal analysis which supports LNT as input language and offers diverse formal techniques like model-checking and simulation.

This paper describes the LNT semantic definition for an AADL model and how its generation and verification are automated using Ocarina and CADP. The remainder of this article is organized as follows: section 2 presents AADL and LNT languages; section 3 gives an overview of the *AADL2LNT* transformation; we describe Ocarina extensions in Section 4; section 5 illustrates our work

[1] AADL is standardized by SAE (Society of Automotive Engineers)
[2] LNT is developed by the Vasy team from INRIA for safety-critical systems
[3] CADP official web site: http://cadp.inria.fr/

with the *Pacemaker* case study; a brief related work is discussed in Section 6; and finally, a conclusion ends the paper in Section 7.

2 BACKGROUND

Briefly, we describe the considered subset of AADL language and we present the LNT language (more details can be found in [19]).

2.1 AADL subset

An AADL model describes a system as a hierarchy of components with their interfaces and their interconnections. AADL components are grouped in three categories: software components (`data`, `subprogram`, `subprogram group`, `thread`, `thread group` and `process`); hardware components (`processor`, `bus`, `virtual processor`, `virtual bus`, `device` and `memory`); and the `system` component. AADL is known by its extensibility, the model can be completed with additional information through properties and annexes. Properties are attached in AADL elements to complete their definitions and bind hierarchically the whole system. Annexes present specific additions, specified with separate syntax, like the `Behavior` annex [2] (BA) for behavioral description.

In our work, the AADL model is viewed as a set of concurrent real-time tasks scheduled by a uniprocessor and asynchronously interacted. Generally described, we consider the following components: `data`, `thread`, `process`, `processor` and `device`. These components are connected through AADL port connections, enriched by the `Behavior` annex, completed with a set of standard properties and finally grouped in the `system` component.

The `thread` component is a schedulable unit that can be executed concurrently with other `threads`. It is declared within a `process` component (represents its virtual address space) and scheduled by the `processor` (bound with *Actual_Processor_Binding* property). `threads` can declare ports that are directional (*in*, *out* or *in out*). They are typed with a `data` component and they may be declared data, event or event data ports. The port connections are explicitly declared between two ports, allowing the transfer of data/event between two components. In addition, `threads` can be completed with the `Behavior` annex. A `Behavior_Specification` clause is included within the `thread` component to specify its behavior as a state-transition system guards and actions.

Note that, in our transformation, a set of temporal and queuing properties should be attributed with these possible values: *Scheduling_Protocol* (RMS, EDF); *Dispatch_Protocol* (periodic, sporadic, timed); *Period*; *Compute_Execution_Time*; *Queue_Processing_Protocol* (FIFO, LIFO); *Overflow_Handling_Protocol* (drop oldest, drop newest); and *Queue_Size*.

2.2 LNT language

An LNT specification represents a system by a set of concurrent processes communicating through channels. The specification distinguishes two parts: a data part defines types and functions; and a control part defines the behavior (process). The control part includes all data part instructions and adds instructions for behavior like asynchronous parallelism and communications.

An LNT specification, typically, consists of a set of LNT modules (in separate source files `*.lnt`) composed of a set of definitions. These definitions may include types, functions, channels and processes using different LNT operators such as variable declaration statement `var`, loops statement `loop`, non-deterministic choice `select`, parallel composition statement `par` and communication. A particular `root process` named `MAIN` should be added to define an entry point of the whole specification.

The LNT specification represents an executable semantic in which all parallel processes start execution and terminate at the same time with the possibility of synchronization. The LNT process is an object describing a behavior. To be synchronized with other processes, the process should declare a set of gates. The communication is ensured through gates typed with channels (gate profiles). The same gate allows sending and receiving messages with a `rendezvous` that blocks the sender until the reception.

3 OVERVIEW OF THE TRANSFORMATION

The *AADL2LNT* transformation is described with a set of corresponding rules between AADL and LNT in a way to obtain a modular LNT specification: every AADL component is transformed and encapsulated into an LNT process. According to component categories, we define a set of LNT generic skeletons as shown in Figure 1 given a global view of the transformation: `SCHEDULER` process implements the `processor` scheduling protocol; `THREAD` process represents the `thread` component with its supported features, subcomponents and behavior specification; and `DEVICE` maps the `device` component. In addition, the AADL port connection is mapped with an auxiliary process `CONNECTOR` using to specify queuing properties and ensure the asynchronism of communications. Finally, all processes are composed and synchronized through LNT gates to form the whole system in the `Main` process.

The *AADL2LNT* transformation can be represented at three levels, allowing the mapping of different kind of AADL models: the **scheduling mapping** ensures the transformation of models with independent `threads` (no port connections); the **communication mapping** completes the mapping by considering port connections between `threads` and `devices`; finally, the `thread` component may be enriched with behavioral descriptions using BA which is transformed at the **behavioral mapping** level. In the rest of this section, we briefly explain each mapping level.

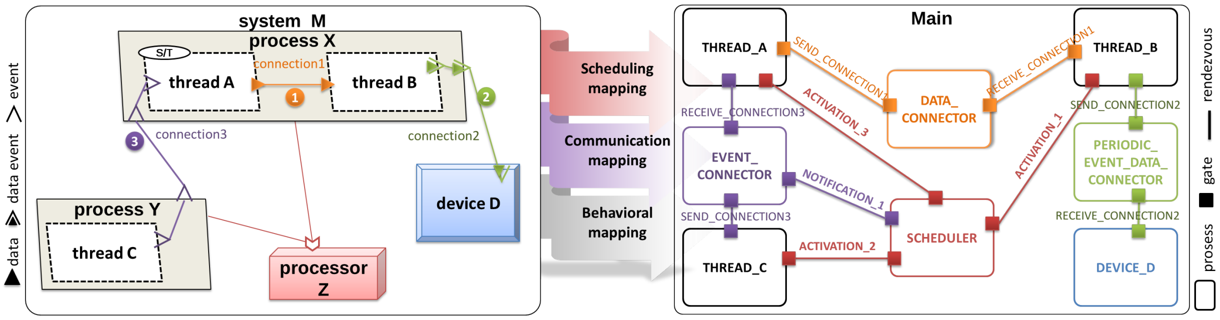
3.1 Scheduling mapping

At this level, we consider only `thread` and `processor` components mapped respectively to `THREAD` and `SCHEDULER` processes.

3.1.1 *Processor mapping.*

The AADL `processor` is a hardware component that ensures the scheduling and execution of `threads`. The `SCHEDULER` process represents the `processor` component by implementing its scheduling algorithm. This process plays a crucial role in the LNT specification, it encapsulates all temporal calculations ensuring the activation of `threads` according to their dispatch protocol. As shown in Figure 1, `SCHEDULER` and `THREADs` are connected by `ACTIVATION` gates. This synchronization allows the `SCHEDULER` to activate `THREADs` using a set of activation orders. Thanks to its programming ability, many scheduling algorithms may be implemented within the `SCHEDULER` process using the LNT language: preemptive/non-preemptive scheduling and fixed/unfixed based-priority scheduling. For example, the RM (Rate Monotonic) scheduler is based on the preemptive fixed-priority

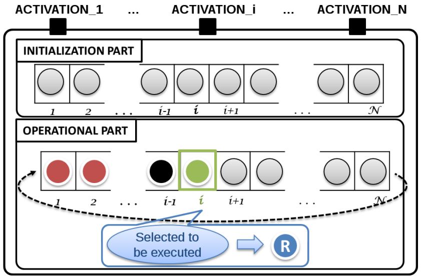
Figure 1: Overview of the *AADL2LNT* transformation



scheduling. In this case, the SCHEDULER uses a set of activation orders distinguishing different alternatives of preemptive scheduling: T_Dispatch_Completion, T_Dispatch_Preemption, T_Preemption and T_Preemption_Completion.

Generally described, the SCHEDULER process consists mainly of two parts as shown Figure 2: **INITIALIZATION** and **OPERATIONAL**. To schedule the threads execution, the SCHEDULER requires the set of threads with their temporal parameters which is included in the **INITIALIZATION PART** using an LNT array (THREAD_ARRAY). These information are extracted, at generation, from the AADL model (attributed by different temporal properties). The **OPERATIONAL PART** is the main part of the SCHEDULER ensuring the execution by applying the scheduling algorithm. The THREAD_ARRAY is visited to select the appropriate thread for execution. The selected thread is handled (allocate its execution time, update its state, prepare its activation order) to send an activation order with its corresponding ACTIVATION gate.

Figure 2: SCHEDULER graphical skeleton



3.1.2 Thread mapping. The THREAD process represents the thread component, it is designed to be scheduled with others THREADs by the SCHEDULER. This process covers an important part of the AADL thread semantics of thread states, dispatching, scheduling and execution described in the AADL standard. The THREAD skeleton is included in Listing 1. It declares the ACTIVATION gate to be synchronized with the SCHEDULER. Its behavior is an infinite loop whose body is a non-deterministic choice select in order to gather different possible values (the considered activation orders) of the ACTIVATION communication.

Listing 1: THREAD LNT skeleton

```
process THREAD_* [ACTIVATION: LNT_Channel_Dispatch] is
    loop select
        ACTIVATION (T_Dispatch_Completion)
        []
        ACTIVATION (T_Dispatch_Preemption)
        []
        ACTIVATION (T_Preemption)
        []
        ACTIVATION (T_Preemption_Completion)
        ...
    end select end loop
end process
```

The THREAD process is dispatched according to the corresponding *Dispatch_Protocol* property. We support periodic, sporadic and timed dispatch models as follows:

- periodic threads are periodically dispatched, at every period (the specified *Period* property value).
- sporadic threads have no fixed first activation, they are activated in response to asynchronous events (invocation-events). The *Period* property value is considered as the minimal delay between two successive activations.
- timed threads are periodically dispatched and may be also activated by invocation-events. The *Period* property value represents a time-out value ensuring that a dispatch occurs after a period if no events have arrived.

The THREAD processes are generically implemented for all dispatch protocols, while the SCHEDULER maintains that: periodic threads are activated with regular-orders; sporadic threads receive irregular-orders according to invocation-events reception; and timed threads are periodically activated as long as there is no invocation-events.

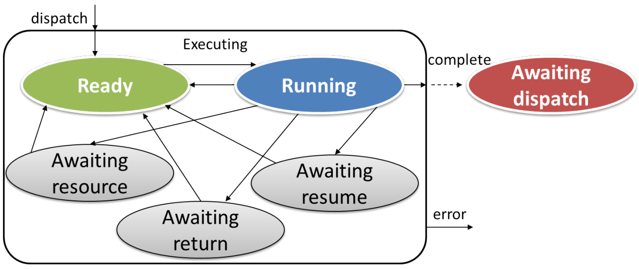
While scheduling, the THREAD may be in one of states defined by the standard *thread scheduling and execution states* automaton added in Figure 3. We consider the Ready state (the thread is able to be executed) and the Running state (the thread is actually executing) and we except awaiting states (shared resources, subprogram calls and background threads) which are not supported in our transformation. A Ready thread can be selected by the SCHEDULER, thus, it moves to the Running state. In addition, while running, the thread can be preempted and so it returns to the Ready state or it can complete its execution and becomes in the Awaiting dispatch which is a suspended state for threads when completing the execution. A thread in Awaiting dispatch state may become Ready by a temporal event (dispatch for a new period) or an invocation-event.

The ACTIVATION communication defines the THREAD states: the current state is defined according to the received SCHEDULER order. Initially, THREAD is supposed in the Ready state. It is suspended until the reception of a new activation order and then it moves to Running state. At this level, four alternatives can be presented:

- T_Dispatch_Complete: THREAD starts and completes the execution of the current period and enters in the Awaiting dispatch state;
- T_Dispatch_Preemption: THREAD starts the execution in the current period but with a preemption, thus, it returns to the Ready state;
- T_Preemption: THREAD progresses in execution without attaining the completion time, so it returns to the Ready state;
- T_Preemption_Complete: THREAD finishes the execution of the current period and enters in the Awaiting dispatch state.

After the execution, THREAD sends an T_Complete to the SCHEDULER meaning that THREAD has accomplished the activation order and it is no more in the Running state.

Figure 3: AADL thread scheduling and execution states [1]



3.2 Communication mapping

This mapping concerns port connections between threads and devices. They are declared at process and system levels and typed with data components. AADL provides a rich semantics of AADL ports and their connections in all aspects (topologies, directions, timing, etc) that can not be totally considered in a single work. Thus, we consider these restrictions: only 1-1 connections; no *in out* ports; event data and event port are similarly handled at this level; inputs are frozen at the start time of execution; and outputs are transferred at the completion time.

In this mapping, we aim to draw thread connections without the consideration of data contents. We simply maps data/event into an enumeration type (AADLDATA label) exchanged between different THREAD through the corresponding channel as included in Listing 2.

Listing 2: data type and channel

```
type LNT_Type_Data is AADLDATA, EMPTY end type
channel LNT_Channel_Port is (LNT_Type_Data) end channel
```

In addition, the device components are supported since they can be connected with threads. devices are not mapped at the **scheduling mapping** level since they are not scheduled, also their internal behavior is not considered. So the device component is simply transformed into a DEVICE process as shown in Figure 1. Mainly, DEVICE declares a set of gates equivalent to the device ports and gathers corresponding communications in its behavior.

3.2.1 Port mapping. The THREAD declaration is completed with the gate declarations corresponding to the thread port declarations as shown in Listing 3. Since we deal with LNT gates, which are bidirectional, *in* and *out* ports are similarly represented. Then, the corresponding communication is also added after the ACTIVATION communication.

Listing 3: THREAD LNT skeleton

```
process THREAD_* [ACTIVATION : LNT_Channel_Dispatch ,
   PORT_* : LNT_Channel_Port] is
   loop select ... [] ...
      ACTIVATION (T_Dispatch_Completion);
      PORT_* (AADLDATA)
      [] ...
   end select end loop
end process
```

3.2.2 Connection mapping. Connections between LNT processes are drawn through gate synchronizations: communications are effected with rendezvous points which block sender until the reception. In this case, the asynchronous aspect of thread connections cannot be presented directly with LNT gate synchronizations. Hence, we add an auxiliary process (CONNECTOR) synchronized between sender and receiver threads.

The CONNECTOR represents the AADL semantic port connection which includes all port connection declarations (at process and system levels) that follow the component containment hierarchy in the instantiated system from a source thread to a destination thread. Thus, all port connection declarations are abstracted in the CONNECTOR synchronizations: connections ❶, ❷ and ❸ of Figure 1 are similarly transformed into CONNECTOR instances.

At THREAD level, communications are also controlled through SCHEDULER orders that fix input and output times:

- T_Dispatch_*: THREAD receives inputs at the start time;
- T_*_Complete: THREAD sends outputs the completion time.

The port type and the thread *Dispatch_Protocol* property allow the definition of three CONNECTOR types as shown in Listing 4. In the case of data port connections and periodic threads, a simple DATA_CONNECTOR is used without buffers. It keeps the data until the next reception, each time a new data is received, the last one is overwritten. When exchanging events, we use EVENT_CONNECTORs including a list of inputs with a definite size (Queue_Size parameter). Event buffers implement the set of queuing properties such as *Queue_Processing_Protocol* and *Dequeue_Protocol*.

Listing 4: Diffrent LNT CONNECTORs

```
process DATA_CONNECTOR
  [INPUT: LNT_Channel_Port, OUTPUT: LNT_Channel_Port]
process PERIODIC_EVENT_CONNECTOR [INPUT: LNT_Channel_Port,
  OUTPUT: LNT_Channel_Port] (Queue_Size : Nat)
process EVENT_CONNECTOR [INPUT: LNT_Channel_Port,
  OUTPUT: LNT_Channel_Port, NOTIFICATION : LNT_Channel_Event]
  (Queue_Size : Nat)
```

Since we support sporadic and timed threads, the SCHEDULER needs to be notified for every new incoming invocation-event: sporadic threads are ignored by the SCHEDULER until the reception of an invocation-event; also timed threads may be activated by invocation-events. So we add NOTIFICATION gates to synchronize

EVENT_CONNECTORs with the SCHEDULER as shown in Figure 1. Thus, when receiving a new event, the EVENT_CONNECTOR notifies the SCHEDULER, to consider the concerned THREAD in scheduling.

3.3 Behavioral mapping

At this level, the AADL model is completed with behavioral descriptions using the Behavior annex. In our work, the BA is mainly supported to complete the thread component with behavior handling inputs and outputs in order to enrich the communication mechanism. The Listing 5 includes the Behavior_Specification which consists of: a *variables* section that allows the declaration of local variables used in descriptions within the scope of the Behavior_Specification clause; a *states* section allowing the declaration of the states of the behavior automaton that may be declared as initial, final or complete state; and a *transitions* section for the description of the behavior by linking states through guarded transitions (S_i-[condition]->S_j {actions}).

A behavior automaton starts from an initial state and terminates in a final state. A complete state, as defined in the BA standard [2], acts as a suspend/resume state out of which threads are dispatched. The transitions specify behavior as a change of the current state from a source state S_i to a destination state S_j. A condition (dispatch or execution) determines whether a transition is taken and then the corresponding actions are performed. The dispatch condition (on dispatch) means that the thread controls its state when it is dispatched periodically or when satisfying the dispatch_trigger_condition which is a boolean expression describing a logical combination of triggering events (the arrival of events on ports). The execution condition signifies that the transition is guarded by a logical expressions based on input values. Periodic threads are always considered to be unconditionally handled by dispatch conditions without the dispatch_trigger_condition. In the case of sporadic or timed threads, invocation-events can be used in the expression of dispatch_trigger_condition, which refines the AADL dispatch model by defining different behaviors for each input when several event ports are declared.

Listing 5: Behavior_Specification

```
annex Behavior_specification {**
variables
 V1 : T1; ... Vm : Tm;
states
 S0 : initial state; Si, Sj : complete state;
 Sl : state;          Sn : final state;
transitions
S0 -[on dispatch dispatch_trigger_condition] -> Si;
Si -[execution_condition]-> Sj {
  -- actions
  };
**};
```

3.3.1 *Data mapping.*

In previous levels, data components were generically mapped into an enumeration type: the consideration of the data contents was useless since they are not handled in the component behavior. Now, inputs and outputs can be used in the BA description and port contents may be exploited in calculations. Thus, the data abstraction should be overridden and the data type should be considered while transformation. Currently, we deal with basic data types: each type from the AADL standard Base_Types package is mapped into a suitable LNT type. In addition, port types are now differently considered: we define for each data, event or event data port a separate LNT channel (Listing 6). The Bool type is used to map event (the *true* value marks a new incoming event).

Listing 6: data new type and channels

```
channel LNT_Channel_Data_Port is (LNT_Type_Data) end channel
channel LNT_Channel_Event_Port is (Bool) end channel
channel LNT_Channel_Event_Data_Port is
  (LNT_Type_Data, Bool) end channel
```

3.3.2 *Behavioral annex mapping.*

Due to the lack of space, we describe briefly this mapping level which needs to be well detailed in a further publications. The general idea consists of completing the THREAD with BA transitions mapping in order to embed the behavior automaton within its state automata: in the Running state, the THREAD may move in one of complete states of the behavior automaton.

As shown in the example of Listing 7, the set of local variables (V_1, .., V_m) are added in the THREAD process using the var statement, they are declared with the same name and the suitable type. All states are added using an enumeration type.

The current state of the behavior automata is explicitly mapped within the THREAD process using a variable STATE. The transitions are specified using the LNT conditional statement (if ((STATE==S_i) and (conditions)) then .. end if), in which, we assign the STATE variable with the new state (STATE:=S_j;) and we include the corresponding actions. For illustration, we give the THREAD_T in Listing 7 whose behavior corresponds to the transition $S0$-[A < 5]-> $S1$ {V1 :=A+1;}.

Listing 7: THREAD LNT skeleton

```
process Thread_T [ACTIVATION: LNT_Channel_Dispatch,
 PORT_A: LNT_Channel_Port] is  -- port declaration
   var
      A : LNT_Type_Data, -- port variable
      V1 : LNT_Type_Data, ... -- behavior variable
      STATE : LNT_Type_STATES -- current state
   in
      STATE := S0;
      A := LNT_Type_Data (0); V1 := LNT_Type_Data (0);
      loop select
         ACTIVATION (T_Dispatch_Completion);
         PORT_A (?A); -- port communication
         if (STATE == S0) and (A < 5) then
            STATE := S1; V1 := A + 1
         end if; ...
```

The on dispatch conditions are assumed implicitly since we include the BA mapping after the ACTIVATION communication and so every new dispatch the THREAD controls its state. The execution conditions or the dispatch_trigger_condition of dispatch conditions are easily added on the LNT if condition of the corresponding transition using logical disjunction and conjunction LNT operators. Since we use LNT Bool type to specify events, the dispatch_trigger_condition is satisfied when its logical expression becomes *true* (new incoming event).

The actions associated with transitions consist of control structures (action sequences, conditionals, finite loops, etc). In our work, basic actions are considered, that may be an assignment action or a communication action. These actions are transformed in the LNT language using the suitable LNT statement such as assignments and gate communications.

4 IMPLEMENTATION

Ocarina[4] [16] is an open source compiler developed since 2004 and recently deployed on GitHub under the OpenAADL project. The Ocarina compiler is designed with a modular architecture. Analyses and generations are handled using ASTs (Abstract Syntax Tree) which are the internal representation of models (AADL, annexes and other languages). Based on the grammar rules, the model is decomposed into a set nodes hierarchically connected to create the corresponding syntax tree. These ASTs are manipulated in three distinguished parts as follows.

- The *Central library* part consists of a set of routines (node builder and finder) allowing the AST construction and manipulation of the AADL models and other languages.
- The *Frontend* part ensures lexical, syntactic and semantic analyses of the AADL model. At this level an AADL AST is created, analyzed and then instantiated (decorated with information). Indeed, if the AADL model includes annexes, they are similarly handled with separate ASTs.
- The *Backend* part provides different automatic generations. The frontend ASTs are expanded and used for model/code generation. Each generation has its specific modules that implement transformation rules to produce code source files of the target language.

In our work, we extend the compiler with a set of modules in its different parts to support the `behavior` annex and the *AADL2LNT* transformation. Briefly, the *Central library* is enriched with the required routines for both BA and LNT ASTs based respectively on BA [2] and LNT [7] grammars. A set of analysis modules is added in the *Frontend* to handle `Behavior_specification` clauses. For each clause, a BA AST is created and hierarchically constructed at the syntax analysis phase. Then, it is attached to the corresponding AADL component. Thus, the AADL AST is completed with BA ASTs to obtain a final AADL-BA AST. Finally, the *AADL2LNT* transformation is implemented in the *Backend* to produce LNT files. In the following, we detail the LNT generation and the obtained tool-chain for AADL model analysis.

4.1 LNT code generation

The *AADL2LNT* is applied on an *instantiable* AADL `system`, means that the model is successfully analyzed lexically, syntactically, semantically and can be completely bound (all components are bounded such as, all `threads` are bound to the `processor`). In addition, a set of standard properties should be attributed (scheduling and queuing properties indicated in section 2.1), otherwise, an error or warning message is displayed. Thus, the AADL AST is created and instantiated to become ready for the LNT generation. Note that depending on AADL model kind, the appropriate transformation level is applied. The non-supported elements (e.g. `bus`, `shared access`) are automatically ignored. And when using `Behavior` annex, the transformation is applied on the complete AADL-BA AST.

4.1.1 Model transformation. Following Ocarina *Backend* architecture, the *AADL2LNT* rules are applied on the AADL AST to simultaneously build the LNT AST. Then, the LNT AST is scanned by the LNT code printer in order to produce code source files (`*.lnt`). To obtain a modular specification, we generate a set of LNT modules which corresponds to the construction of a set of LNT LTSs. Two main modules, **_Types* and **_Main*, are always generated for all LNT specifications. Then, according to mapping level, a set of LNT modules is added as follows:

- **_Threads* module consists of a set of `THREAD` and `DEVICE` declarations, whose generation depends mainly on port declarations in `thread` or `device` components.
- **_Processor* module contains the `SCHEDULER` process with a set of LNT function definitions required for scheduling and execution. The `SCHEDULER` generation needs the extraction and the calculation of a set of `thread` information to be included in its **INITIALIZATION PART**: the `THREAD_ARRAY` generation requires the number of `thread` instances and the set of values of each `thread` properties.
- *Port_Connections* and *Port_Connections_BA* are generic modules included in the compiler resources. Each module consists of a set of `CONNECTOR` declarations that will be instantiated for each port connection.

All declared processes in different modules should be instantiated and synchronized to form the whole system in the `Main` process which requires the generation of two modules:

- **_Types* module consists of a set of LNT types and channels required for different synchronizations: `THREAD/CONNECTOR`, `CONNECTOR/SCHEDULER` and `THREAD/SCHEDULER`.
- **_Main* module contains the `Main` process whose generation can be resumed in three steps:
  (1) preparation of the list of `thread` instances;
  (2) synchronization of `CONNECTOR` processes: for each port connection, we create one `CONNECTOR` instance which is synchronized between `THREADs` equivalent to the source *in* port and the destination *out* port `threads`;
  (3) global composition: all `THREADs` are synchronized with the `SCHEDULER` on `ACTIVATION` gates, all `CONNECTORs` are synchronized with the `SCHEDULER` on `NOTIFICATION` gates.

To provide a traceable transformation, a set of naming rules is applied while generation. At the `THREAD` level, all AADL port identifiers are conserved and prefixed by `"PORT_"`. Also, all port content variable, BA variable and state identifiers are conserved while the **behavioral mapping**. At the `Main` level, the generation conserves AADL component identifiers as follows: the `thread` identifier is prefixed by `"THREAD_"`; the `device` identifier is prefixed by `"DEVICE_"`; for each AADL connection, two identifiers are prepared (the connection name is prefixed by `"SEND_"` and `"RECEIVE_"`) to present two gates for the `THREAD/CONNECTOR` synchronization.

4.1.2 SVL script. In addition to LNT modules, a second input is necessary for analysis with CADP toolbox. A script file (`demo.svl`) containing a set of operations with SVL (Script Verification Language) [10] is constructed to orchestrate the verification. This file is generated directly for each AADL system (without an SVL AST) containing two parts: compilation of the LNT specification and a set of generic properties for verification by model-checking (deadlock, schedulability, data loss, buffer overflow). The model-checking technique consists of checking if a model satisfies a given property

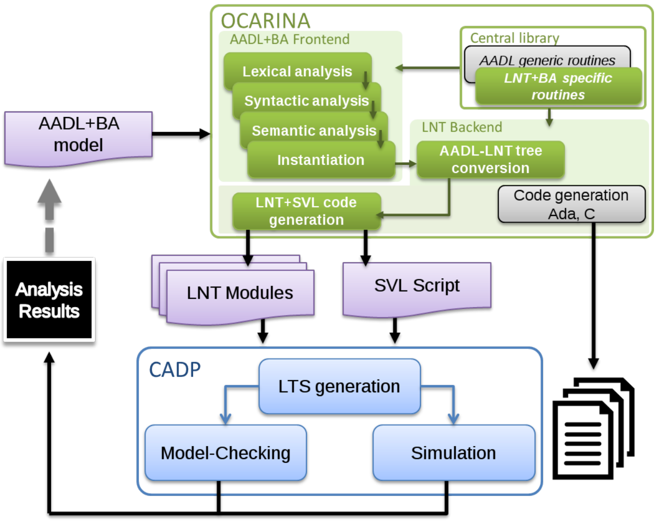
[4] https://github.com/OpenAADL/ocarina

specified with temporal logic. In our work, properties are specified by the SVL `property` statement, that embeds the temporal logic verification statement, and checked with the CADP *Evaluator* model-checkers [17, 18].

4.2 Tool-chain

The *AADL2LNT* transformation allows the definition of a tool-chain, depicted in Figure 4, using Ocarina for architectural modeling and CADP for formal verification.

Figure 4: Ocarina-CADP tool-chain



The provided tool-chain ensures a transparent model transformation and verification. The transformation is proceeded using Ocarina command line, then, the generated SVL script is simply invoked to begin model analysis. Initially, the LNT specification is transformed into an LTS (Labeled transition system), to be exploited by verification tools. Then, the system is automatically checked by a set of generic properties, and it can be simulated with CADP simulators like OCIS [9].

The analysis results help designers in model correction and improvement (validation of temporal parameters, deadlock detection, schedulability test, detection of connection failures, etc). This verification may be iteratively applied after each model modification until the generation of the final application. Moreover, the obtained specification can be considered as a good base of a manual verification. Designers, with formal expertise, may complete different LNT modules and verify new properties for a specific case study.

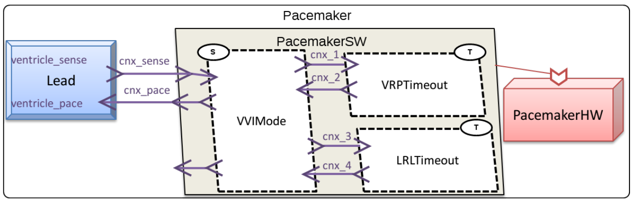
5 CASE STUDY

The current Ocarina version covers almost the BA frontend analysis and the generation of the **scheduling** and **communication mapping**, while for the **behavioral mapping**, further work needs to be completed to include the BA mapping within the `THREAD` skeleton. In this section, we report experiments performed on the *Pacemaker* case study. The *Pacemaker* is a medical device inserted in the body of a patient, to regulate his/her heart beating with electrical impulses. It is used in the case of heart rhythm problems (inability to maintain a normal heart rate). Depending on the heart problem, the *Pacemaker* provides different modes that perform different kinds of therapeutic behavior. In this paper, we consider the VVI mode which is the heart ventricle-chamber pacing mode.

5.1 Ocarina: modeling and generation

AADL modeling. The *Pacemaker* AADL model [5] [15] is depicted in Figure 5. Structurally speaking, the AADL system consists of two main parts: the pulse generator (`PacemakerSW process`) which embeds the software implementing the therapeutic behavior to monitor leads (`Lead device`) for ventricle-chamber. The pulse generator is connected with leads through event ports to sense the heart beating (`sense` *in* port) and send an order for pacing (`pace` *out* port). The VVI therapeutic mode is accomplished using three `threads` interconnected and enriched with BA specifications: a sporadic `thread` to present the VVI mode (`thread VVIMode`); and two timed `threads` (`VRPTimeout`/`LRLTimeout`) representing timers. These `threads` are collaborated to maintain the rate of pacing as follows: when the heart has no beat during an LRL period, `VVIMode` causes a `pace` (an event on `pace` port); if the sense (events on `sense` port) comes too soon after a beat (during the VRL period), it will be ignored; and when the heart is beating regularly, `VVIMode` detects a normal beating rhythm (an event on `normal` port). This behavior is mainly described within the `VVIMode thread Behavior_Specification` included in Listing 8.

Figure 5: AADL model of the *Pacemaker* case study



Listing 8: The `VVIMode` behavior specification

```
annex Behavior_Specification {**
 states
  s1 : initial complete final state;
 transitions
  s1-[on dispatch vrp_timeout]-> s1 {vrp := 0 };
  s1-[on dispatch sense]-> s1 {
   if (vrp=0) normal!; p_or_n_vrp!; p_or_n_lrl!; vrp := 1 end if};
  s1-[on dispatch lrl_timeout]-> s1 {
   pace!; p_or_n_vrp!; p_or_n_lrl!; vrp := 1 };
**};
```

LNT generation. The *Pacemaker* LNT specification with its SVL script are generated by the *AADL2LNT* extension. In Listing 9, we include an extract from the obtained process `THREAD_VVIMode` equivalent to the `VVIMode thread`. The process gathers mainly the set of the equivalent declarations (gates and variables), the set of the required initializations and the BA code inserted after the `ACTIVATION` communication.

[5] This model is inspired from the pacemaker model published by Ellidiss technologies.

Listing 9: An extract of the THREAD_VVIMode

```
process THREAD_VVIMode [ACTIVATION: LNT_Channel_Dispatch,
 PORT_SENSE:LNT_Channel_Event, .. , PORT_PACE:LNT_Channel_Event] is
var
   STATE : LNT_Type_STATES,
   VRP : Nat, ...
in
   STATE := S1; VRP_TIMEOUT := false; ...
   loop
      ACTIVATION (T_Dispatch_Completion);
      PORT_SENSE (?SENSE);
      PORT_LRL_TIMEOUT (?LRL_TIMEOUT);
      PORT_VRP_TIMEOUT (?VRP_TIMEOUT);
      if (STATE == S1) and (VRP_TIMEOUT) then
         STATE := S1; VRP := 0
      elsif (STATE == S1) and (LRL_TIMEOUT) then
         STATE := S1; PORT_PACE (true);
         PORT_P_OR_N_VRP (true); PORT_P_OR_N_LRL (true);
         VRP := 1
  ... end if; ...
```

The Table 1 resumes the *Pacemaker* transformation metrics. The obtained specification counts 765 lines, whose 75% is automatically generated by the current Ocarina version. This generation covers an important part of *AADL2LNT* transformation and eliminates the complexity of its rules. Particularly, the Main mapping seems tricky since we should compose a lot of process instances through an important number of gates: the *Pacemaker* Main process counts 12 instances synchronized on 23 different gates. The obtained specification complexity depends directly of threads/connections type and number: from models with independent threads, we obtain simple LNT models without CONNECTOR instances. While with highly connected models, especially for sporadic or timed threads, the process number increases significantly. Despite this fact, the obtained specifications still comprehensible and the correspondence between the AADL/BA syntax and the LNT code can be easily identified thanks to the applied naming rules (section 4.1.1).

Table 1: *Pacemaker* metrics

	AADL	LNT	SVL
Number of	lines	lines	lines
	157	765	40
Equivalence of	element	process	
	2 thread	2 THREAD	
	1 process	-	
	1 device	1 DEVICE	
	15 event connections	7 EVENT_CONNECTOR	
	1 processor	1 SCHEDULER	
	1 system	1 Main	

5.2 CADP: formal analysis and evaluation

LTS generation. The obtained LNT specification is ready for CADP analysis. The demo.svl script allows the use of CADP tools for LTS generation and model-checking of a set of properties. The LTS represents the state space of the obtained LNT specification, characterized by a number of states and transitions. In the formal context, a well known problem is the state space explosion, when the number of states or transitions explodes. For this reason, the definition of the *AADL2LNT* transformation was refined several times (since the first proposition [19]) to obtain the smallest state spaces. The *Pacemaker* LTS counts 5473 states and 5473 transitions which present a small state space compared to the considered AADL model with sporadic/timed threads, event port connections and BA descriptions. We note that with independent or only-periodic thread models, we obtain much smaller spaces. In more advanced experiments, we test excessively models with connected sporadic/periodic threads. We note that AADL models can reach 40 threads without space explosion problem (state spaces are about 900 states for models without BA descriptions).

Model-checking. After the LTS generation, we can move to model-check the set of generated properties. For illustration, Listing 10 presents the Connection property that assumes that every connection is well established: for each event sent, there is an inevitably reception. As shown in Listing 10, SVL property statements can be parameterized. Parameters are used to present connections and threads, so properties are separately applied on each thread/port connection. Thus, we obtain understandable results and the AADL model problems can be rapidly localized. For example, the Connection property is checked for each connection (the model-checker displays a true/false response): the Listing 11 presents the result of the verification of the inter-thread connection CNX_3 from the *Pacemaker* AADL model.

Listing 10: An extract of the *Pacemaker* SVL script

```
-- LTS generation
"Main.bcg"= divbranching reduction of "PACEMAKER_Main.lnt";
-- Properties for model-checking
property Connection (ID)
 "After a SNED action, a RECEIVE is eventually reachable" is
 "Main.bcg" |= with evaluator4
 AFTER_1_INEVITABLE_2 (SEND_$ID, RECEIVE_$ID); expected TRUE;
end property;
check Connection (CNX_PACE); check Connection (CNX_3);
```

Listing 11: Analysis results from *Pacemaker* case study

```
property Connection (CNX_3)
  | After a SNED action, a RECEIVE is eventually reachable
PASS
```

More analysis. The *Pacemaker* model was successfully analyzed with different generated properties. In addition, advanced analysis was applied to verify the VVI therapeutic behavior of the pulse generator. As illustration, the normal rhythm detection can be checked as follows: the DEVICE_Lead is completed to be synchronized with the SCHEDULER, and so it can be periodically dispatched to send events at a normal heart rate on the ventricle_sense port; and a new SVL property (Listing 12) is added to check if the VVIMode thread sends events on the normal port without any pacing event.

Listing 12: Normal_Rhythm SVL property

```
 property Normal_Rhythm is "Main.bcg" |= with evaluator3
  NEVER ("PACEMAKERSW_PACE__LEAD_VENTRICLE_PACE !TRUE") and
  [true*."SEND_PACEMAKERSW_NORMAL__LEAD_NORMAL !TRUE"]true;
  expected TRUE;
end property;
```

6 RELATED WORK

The AADL language is supported by several tools, either open-source (OSATE, Ocarina, TOPCASED) or commercial (STOOD, AADL Inspector). In our work, we contribute on Ocarina open-source tool suite that can be used as stand-alone compiler since it provides different engineering steps (modeling, analysis and code generation) with the possibility of the use of extra tools like Cheddar and Bound-T. In addition, Ocarina can be easily integrated as a backend for other AADL editors (already used through OSATE and AADL Inspector tools), which increases the visibility of our work.

AADL Formal approaches are often based on model transformation into different languages such as Lustre [13], TLA+ [22], Signal [5], ACSR [24], TASM [25], Fiacre [4], Real-time Maude [20] and BIP [8]. As examples: many work [5, 26] are around the Polychrony platform and Signal language for synchronous verification of AADL models where behavior is specified by BA or Simulink. The transformations are implemented within OSATE and the analyses are performed by Polychrony and SynDEx tools; authors in [25] transform a synchronous AADL subset (periodic threads, data port connections, modes and BA) to the TASM language, in order to verify behavioral properties (deadlock and reachability) with TASM tool suite and UPPAAL; the transformation of AADL model into the Fiacre language is addressed in the TOPCASED environment [4]. This work supports periodic/sporadic thread and event/data port connections, but it is restricted to the non-preemptive scheduling. The AADL model needs a first transformation into Fiacre which is considered as a pivot language for other transformations for connection to Tina and CADP tools. This proposition is extended in [6] aiming the validation of the transformation for an AADL synchronous subset; authors in [20] use the Real time Maude language to transform an AADL subset with BA to be analyzed with the Real-Time Maude platform. This work is extended in [3] considering a defined Synchronous AADL sub-language.

Generally speaking, these transformations aim at the same objectives: the definition of an AADL executable formal semantics for model-checking of behavioral and temporal properties. Yet, they can be distinguished according to the considered AADL subset (synchronous subsets, non-preemptive scheduling, etc). We note also that most of these transformations is implemented as OSATE plugins and a majority requires more than one model transformation to be connected to the analysis tools. Comparing to these work, we provide a tool-chain that automatizes both generation and model-checking. We use a direct input language to CADP without additional transformations. We consider an important AADL subset that especially includes event-driven threads, asynchronous connections and preemptive scheduling. The considered subset covers fundamental real-time features that can be used in more realistic applications rather than synchronous and non-preemptive approaches.

7 CONCLUSION AND FUTURE WORK

In this paper, we described the *AADL2LNT* Ocarina extension for AADL formal semantic generation in the context of software engineering of real-time systems. This work is based on the transformation of the AADL model into an LNT specification considering mainly scheduling execution and task communication which are indispensable features for a useful real-time analysis. In addition, we support the Behavior annex for more complete AADL models. We provide an automatic tool-chain for formal analysis of AADL models by CADP toolbox, through which the LNT specification is generated and then checked by a set of generic properties.

The current *AADL2LNT* version allows an important generation and we are working continuously on our implementations in Ocarina for a total automation. As future work, we aim to extend the **scheduling mapping** to support multi-core scheduling.

REFERENCES

[1] 2009. AS5506A:SAE Architecture Analysis & Design Language (AADL) Ver 2.0.

[2] 2011. AS5506/2:SAE Architecture Analysis & Design Language, Annex Vol 2.

[3] K Bae, P C Ölveczky, and J Meseguer. 2014. Definition, semantics, and analysis of Multirate Synchronous AADL. In *International Symposium on Formal Methods*. Springer, 94–109.

[4] B Berthomieu, J P Bodeveix, C Chaudet, S Dal Zilio, M Filali, and F Vernadat. 2009. Formal verification of AADL specifications in the Topcased environment. In *International Conference on Reliable Software Technologies*. Springer, 207–221.

[5] L Besnard, A Bouakaz, T Gautier, P Le Guernic, Y Ma, J P Talpin, and H Yu. 2015. Timed behavioural modelling and affine scheduling of embedded software architectures in the AADL using Polychrony. *Science of Computer Programming* 106 (2015), 54–77.

[6] J P Bodeveix, M Filali, M Garnacho, R Spadotti, and Z Yang. 2015. Towards a verified transformation from AADL to the formal component-based language FIACRE. *Science of Computer Programming* 106 (2015), 30–53.

[7] D Champelovier, X Clerc, H Garavel, Y Guerte, C McKinty, V Powazny, F Lang, W Serwe, and G Smeding. 2016. Reference manual of the LNT to LOTOS translator (version 6.5). *Inria/Vasy and Inria/Convecs* 143 (2016).

[8] M Y Chkouri, A Robert, M Bozga, and J Sifakis. 2009. Translating AADL into BIP-application to the verification of real-time systems. In *Models in Software Engineering*. Springer, 5–19.

[9] H Garavel. 1998. Open/Cæsar: An open software architecture for verification, simulation, and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 68–84.

[10] H Garavel and F Lang. 2002. SVL: a scripting language for compositional verification. In *Formal Techniques for Networked and Distributed Systems*. Springer, 377–392.

[11] H Garavel, F Lang, R Mateescu, and W Serwe. 2013. CADP 2011: a toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer* 15, 2 (2013), 89–107.

[12] M E Hamdane, A Chaoui, and M Strecker. 2013. From AADL to Timed Automaton-A Verification Approach. *International Journal of Software Engineering & Its Applications* 7, 4 (2013).

[13] E Jahier, N Halbwachs, P Raymond, X Nicollin, and D Lesens. 2007. Virtual execution of AADL models via a translation into synchronous programs. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*. ACM, 134–143.

[14] A Johnsen, K Lundqvist, P Pettersson, and O Jaradat. 2012. Automated Verification of AADL-Specifications Using UPPAAL. In *HASE*. 130–138.

[15] B R Larson. 2014. Formal semantics for the pacemaker system specification. *ACM SIGAda Ada Letters* 34, 3 (2014), 47–60.

[16] G Lasnier, B Zalila, L Pautet, and J Hugues. 2009. Ocarina: An environment for AADL models analysis and automatic code generation for high integrity applications. In *Reliable Software Technologies–Ada-Europe*. Springer, 237–250.

[17] R Mateescu and M Sighireanu. 2003. Efficient on-the-fly model-checking for Regular Alternation-free mu-Calculus. *Science of Computer Programming* 46, 3 (2003), 255–281.

[18] R Mateescu and D Thivolle. 2008. A model checking language for concurrent value-passing systems. In *International Symposium on Formal Methods*. Springer, 148–164.

[19] Hana Mkaouar, Bechir Zalila, Jérôme Hugues, and Mohamed Jmaiel. 2015. From AADL Model to LNT Specification. In *Reliable Software Technologies–Ada-Europe 2015*. Springer, 146–161.

[20] P C Ölveczky, A Boronat, and J Meseguer. 2010. Formal semantics and analysis of behavioral AADL models in Real-Time Maude. In *Formal Techniques for Distributed Systems*. Springer, 47–62.

[21] X Renault, F Kordon, and J Hugues. 2009. From AADL architectural models to Petri Nets: Checking model viability. In *IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE, 313–320.

[22] J F Rolland, J P Bodeveix, M Filali, D Chemouil, and D Thomas. 2008. Modes in asynchronous systems. In *Engineering of Complex Computer Systems, ICECCS*. IEEE, 282–287.

[23] A E Rugina, K Kanoun, and M Kaâniche. 2008. The ADAPT Tool: From AADL Architectural Models to Stochastic Petri Nets through Model Transformation. *CoRR* abs/0809.4108 (2008).

[24] O Sokolsky, I Lee, and D Clarke. 2006. Schedulability analysis of AADL models. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 8–pp.

[25] Z Yang, K Hu, J P Bodeveix, L Pi, D Ma, and J P Talpin. 2011. Two Formal Semantics of a Subset of the AADL. In *Engineering of Complex Computer Systems (ICECCS)*. IEEE, 344–349.

[26] H Yu, Y Ma, T Gautier, L Besnard, J P Talpin, P Le Guernic, and Y Sorel. 2013. Exploring system architectures in AADL via Polychrony and SynDEx. *Frontiers of Computer Science* 7, 5 (2013), 627–649.