International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-019-00513-7

REGULAR PAPER

A formal approach to AADL model-based software engineering

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

Abstract
Formal methods have become a recommended practice in safety-critical software engineering. To be formally verified, a system should be specified with a specific formalism such as Petri nets, automata and process algebras, which requires a formal expertise and may become complex especially with large systems. In this paper, we report our experience in the formal verification of safety-critical real-time systems. We propose a formal mapping for a real-time task model using the LNT language, and we describe how it is used for the integration of a formal verification phase in an AADL model-based development process. We focus on real-time systems with event-driven tasks, asynchronous communication and preemptive fixed-priority scheduling. We provide a complete tool-chain for the automatic model transformation and formal verification of AADL models. Experimentation illustrates our results with the *Flight control system* and *Line follower robot* case studies.

Keywords Safety-critical software engineering · Real-time systems · Ravenscar profile · AADL · Formal specification · Model-checking · CADP

1 Introduction

Software engineering in safety-critical domains such as transport, health and aerospace industries is a quite delicate field in computer science. In such a context, designers often cope with large distributed, real-time and embedded systems with diverse requirements. Several approaches (modeling, verification, code generation and testing) have focused on simplifying such complex constructions with more abstraction in system design and automation in development tool-chains [56]. Among these approaches, we note the Model-Driven Engineering (MDE) methodology. It is a development trend based on modeling language, model transformation, production of documentation and code generation. Theoretically, MDE approaches aim to abstract the system representations and allow a coherent evaluation of the system from the specification until the final application. Technically, tools supporting MDE provide mainly automatic generation, analysis and simulation of models and code.

Other promising approaches are formal methods, which refer to mathematically rigorous techniques and tools for the specification and verification of systems. During the past decades, formal methods have become one of the advocated techniques in safety-critical software engineering [56]. Indeed, they are now accepted in certification processes (e.g., DO-333 [51], formal methods supplement to DO-178C and DO-278A standards for avionics systems) as a way to get certification credits by authorities. For these reasons, the integration of formal methods in MDE approaches seems rewarding.

However, their application requires a formal expertise: the considered system should be specified with a specific formalism such as automata and Petri nets, based on a formal semantics described using mathematical approaches, to be explored by dedicated analysis tools. In contrary, the seman-

✉ Hana Mkaouar
hana.mkaouar@redcad.org

Bechir Zalila
bechir.zalila@redcad.org

Jérôme Hugues
jerome.hugues@isae.fr

Mohamed Jmaiel
mohamed.jmaiel@redcad.org

[1] ReDCAD Laboratory, University of Sfax, National School of Engineers of Sfax, BP 1173, 3038 Sfax, Tunisia

[2] Institut Supérieur de l'Aéronautique et de l'Espace (ISAE-SUPAERO), Université de Toulouse, 31055 Toulouse Cedex 4, France

[3] Digital Research Center of Sfax, B.P. 275, 3021 Sakiet Ezzit, Sfax, Tunisia

Published online: 07 March 2019

tics of modeling (architectural) languages such as AADL and MARTE is often given in natural language (i.e., standard and manual documents). This lack of formal semantics makes modeling languages inappropriate for formal verification, they cannot be explored directly by formal analysis tools. Therefore, it is useful to provide an automatic model generation of the formal specification that can be used and reused to encourage the practice of formal methods and assist designers in system verification.

In this context, we aim to integrate a formal verification phase in an MDE approach based on the AADL (Architecture Analysis and Design Language) [1] language. AADL is an industrial architectural language for critical domains such as avionics, automotive electronics and robotics. It is considered as a leader in real-time modeling and ranked in [39] among the top-used languages in industry. AADL is standardized by the SAE[1], and its second version was published in 2009 and revised in 2016.

Our contribution consists of two main parts. In the first one, we describe and justify a formal mapping of a real-time task model compliant with the Ravenscar profile [13] for safety-critical systems. This mapping is based on a conventional tasking model inspired from Liu and Layland [37] with rigorous semantics and strong requirements defined by the Ravenscar profile [13]. It is designed to be modular and comprehensible so that it can be easily extended and used in MDE approaches. We mainly support periodic and sporadic tasks, which are asynchronously connected and concurrently executed by a preemptive fixed-priority scheduler. Different model features are specified with the LNT[2] [14,23] formal language, which is a process algebra based on two standards LOTOS and E-LOTOS [23]. LNT provides sufficiently expressive operators for data and behavior with user-friendly notations to simplify writing and extension. The LNT language is supported by the CADP [22] (Construction and Analysis of Distributed Processes) toolbox. This analysis tool, developed since the mid-1980s, offers diverse formal methods like model-checking and simulation. It is a well-experimented toolbox, used in many industrial applications (e.g., Airbus company [19]).

In the second part, we describe an AADL MDE approach integrating the formal verification at the modeling phase. This allows the early detection of deeper problems that can lead to serious errors in the final application. Especially for real-time systems that consider both concurrency and real-time requirements, it is necessary to validate temporal parameters such as periods, capacities, scheduling protocols, etc. This verification is supposed to be automatic and transparent to simplify and encourage the practice of formal methods in software engineering.

For AADL formal verification, existing works often adopt the transformation of AADL models into other formal models such as timed automata [34], BIP [15], TASM [57] and FIACE [7], to allow their formal verification with existing analysis tools like Tina, UPPAAL and Polychrony. In general, the focus of these approaches is on the model transformation to generate a formal specification, without automating the next steps required to succeed the model verification. Indeed, providing the formal specification is an important step, but further steps are required to accomplish the verification. The formal techniques are applied on the state space of the system, built from the formal specification according to the language semantics. The properties (system requirements to be verified) should be also specified as graphs or temporal logic properties using dedicated formalisms. In addition, many existing AADL transformations ignore important issues concerning the generated analysis results (utility, usability, etc.), and the fact that the formal verification may fail because of scalability problems.

In our work, we propose and evaluate a transformation *AADL2LNT* from the AADL language to the LNT language. A basic proposal was discussed in [43], we presented a transformation from AADL into LNT about only periodic tasks. In this paper, we propose to optimize and complete the LNT mapping to support a Ravenscar compliant task model. The *AADL2LNT* transformation is implemented within the Ocarina [25] tool suite, a development environment for AADL modeling, in order to automatically generate an LNT specification ready for analyzing with the CADP toolbox. In addition, a script file, written in the SVL (Script Verification Language) [20] language, is also generated to assist the verification with CADP. This script ensures the generation of the state space of the LNT specification and the verification of a set of generic properties to check serious problems such as the deadlock detection, the schedulability test and the detection of connection failures. Finally, the verification phase ends with the generation of user-friendly analysis results, easily interpreted by non-formal-expert designers. As part of our experiment, a scalability study is carried out to show the effectiveness of the proposed solution in the case of large systems.

The remainder of this article is organized as follows: Sect. 2 presents some elements of both LNT and AADL languages required in our contribution. In Sect. 3, we develop an LNT formal mapping for a real-time task model. Section 4 details the *AADL2LNT* transformation. Experimental results are discussed in Sect. 5. Section 6 discusses related work. Finally, conclusions and future work end the paper in Sect. 7.

[1] Society of Automotive Engineers.

[2] LNT is developed by the VASY and CONVECS teams from Inria for safety-critical systems.

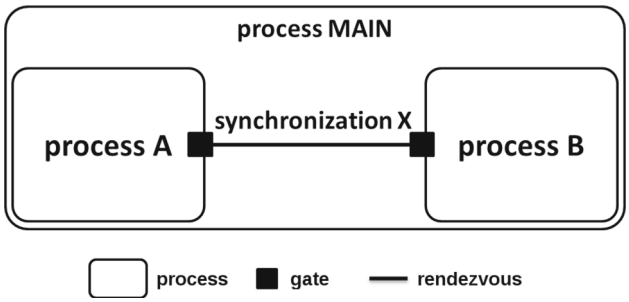

Fig. 1 LNT graphical representation

2 Background

In this section, we present the LNT language. Then, we briefly introduce a subset of the AADL language considered in our MDE application.

2.1 LNT

The LNT [14] language combines features from process algebras and programming languages with a dynamic semantics based on the formal SOS (Structural Operational Semantics) rules.

The LNT specification distinguishes two parts: a data part that defines types and functions; and a control part that defines the behavior (within processes). The data part is a fully imperative language in syntax and semantics. The control part includes almost all data part and adds constructs for behavior like non-deterministic choice, process parallelism and communication.

In the rest of this section, we informally present some definitions of the LNT language (partially represented in Fig. 1). We adopt these identifiers following the LNT reference manual [14] notations: M for module, Π for process, B for behavior, I for statement, T for type, G for gate and Γ for channel.

Module The LNT specification typically consists of a set of LNT modules. An LNT module M (Listing 1) is named with the same name as its source file (`*.lnt`). M can import other modules (M_0,..., M_n); thus, all the definitions of M_0, ..., M_n are visible and can be used in M definitions. These definitions include the LNT types, channels, functions and processes.

Listing 1 LNT module definition

```
module M [(M_0, ..., M_n)] is
    definition_0 ... definition_q
end module
```

Type The LNT language provides a set of predefined basic types (Boolean, Natural, Integer, Real, Character and String) with the associated predefined functions (basic operations such as addition and comparison) that are automatically available. In addition, the language allows the definition of non-basic types (Listing 2). A non-basic type may be a list, a sorted list, a set, an enumerated or a range type (specified within the `type_expression`, as illustrated with the T_{exp} type).

Listing 2 LNT type definition

```
type T is
    type_expression
end type
type T_exp is -- example
    list of BOOL
end type
```

Channel An LNT channel (Listing 3) is a gate type that fixes the types of values to be sent or received during the communication on a given gate. The channel may use user-defined or predefined basic types, as illustrated with the Γ_{exp} channel.

Listing 3 LNT channel definition

```
channel Γ is
    (T_1,1 , ..., T_1,n) , ..., (T_m,1 , ...,
      T_m,l) ,
end channel
channel Γ_exp is -- example
    T_exp
end channel
```

Process An LNT process (Listing 4) is an object describing a behavior. It can be parametrized with a list of formal gates, variables and exceptions. The behavior B comprises sequential composition, conditional behavior (`if`), variable declaration statement (`var`), loop behavior (`loop`), non-deterministic choice (`select`), parallel composition (`par`), communication, etc.

Listing 4 LNT process definition

```
process Π [gate_declaration_0 , ..., gate_declaration_m]
          (formal_parameter_0 , ..., formal_parameter_n)
is
    B
end process
```

Communication The LNT processes can be in communication through the gates and channels (Listing 5). Each process has a set of gate declarations to be synchronized with other processes. The same gate allows sending and receiving messages, compatible with a channel, with a `rendezvous` that blocks the sender until the reception.

Listing 5 LNT Communication

```
process Π [G_0 : Γ_k , ..., G_m : Γ_l] is
    G_i (!V) -- output of expression V
    G_i (?P) -- input in pattern P
end process
```

Non-deterministic choice An LNT `select` statement (Listing 6) allows a non-deterministic choice between behaviors $B_0, \ldots, B_n$.

Listing 6 LNT select statement

```
select
    B0  []  Bi  []  Bn
end select
```

Variable declaration An LNT `var` statement (Listing 7) allows the definition of local variables with their names and their types. The scope of each variable is the statement I, in which it can be assigned values multiple times.

Listing 7 LNT var statement

```
var
    variable_declaration0 , ... , variable_declarationn
in
    I
end var
```

Parallel composition An LNT `par` statement (Listing 8) is used for behaviors $(B_0, \ldots, B_m)$ parallelism and gates $(G_0, \ldots, G_k$ and $G_{(i,0)}, \ldots, G_{(i,n_i)})$ synchronization. The behavior B_i often represents a process instantiation $(\Pi\ [G_0, \ldots, G_k, G_{(i,0)}, \ldots, G_{(i,n_i)}])$. The `par` composition allows two types of synchronization: global and interface. The global synchronization is defined by $G_0, \ldots, G_k$, this communication can happen only if all processes can make it simultaneously. The interface synchronization is defined by $G_{(i,0)}, \ldots, G_{(i,n_i)}$. In this case, if a process is waiting for a communication on a gate which belongs to its synchronization interface (e.g., $G_{(i,j)}$), this communication can happen only if all processes synchronized on the same gate (contain $G_{(i,j)}$ in their synchronization interface) can make it simultaneously.

Listing 8 LNT parallel composition

```
par [G0 , ... , Gk in]
    [G(0,0) , ... , G(0,n0) -> ] B0
    ||
    [G(i,0) , ... , G(i,ni) -> ] Bi
    ||
    [G(m,0) , ... , G(m,nm) -> ] Bm
end par
```

`MAIN` process In the LNT specification, the system is represented by a set of concurrent processes communicating through gates typed with channels. As shown in Fig. 1, a `root process` named `MAIN` should be added to define an entry point of the whole specification. The specification represents an executable semantics in which all parallel processes start execution and terminate at the same time with the possibility of synchronization by *rendezvous*.

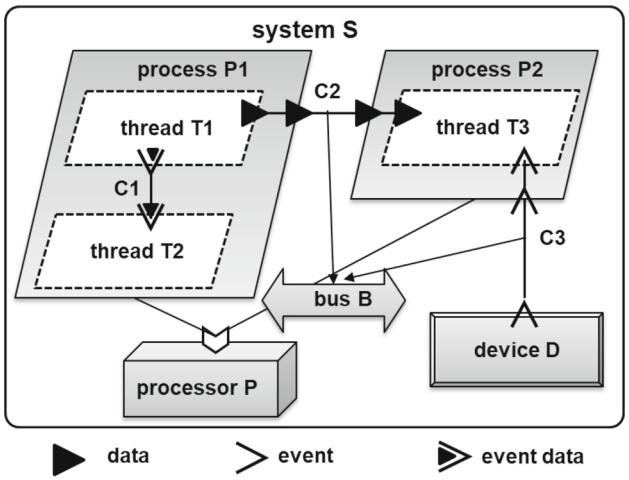

Fig. 2 AADL `system` graphical representation

2.2 AADL

The AADL [1,18] language is an architecture description language for model-based engineering of embedded real-time systems. It defines notations, expressed using both a graphical and a textual syntax, to represent a full system with its software and hardware components in one model (architectural representation of the system).

As shown in Fig. 2, the core language describes a system as a hierarchy of components with their interfaces and connections. The AADL components are defined by a type (contains mainly the component interface elements called features) and zero or more implementations (present the component internal structure composed of subcomponents, calls, connections, flows, modes and properties). The components are grouped in three categories: software components (`subprogram`, `subprogram group`, `data`, `thread`, `thread group` and `process`); hardware components (`processor`, `virtual processor`, `device`, `bus`, `virtual bus` and `memory`); and system composition component (`system`). An AADL connection is a linkage established between components that can be port, parameter or access connections.

The AADL language brings the capability to enrich the model with additional information by a set of standard properties and annexes. Properties are used to complete the component definition and bind the whole system hierarchically. Big and specific additions are specified by separate annexes such as the `Error-Model` annex to specify the fault behavior/propagation, the `ARINC653` annex for the avionics modeling and the `Behavior` annex [2] to specify the architectural behavior.

In the rest of this section, we briefly present some AADL elements (graphically represented in Fig. 2) that are supported in our work.[3]

Thread A `thread` component is a concurrent schedulable unit of sequential execution code. It should be declared within a `process` component, and it is bound to a `processor` component to be scheduled.

Process A `process` component is a virtual address space containing `data`, `thread` and `subprogram` associated with the `process` and its subcomponents.

Processor A `processor` component is an abstraction of hardware and software for the scheduling and execution of `threads`.

Device A `device` component is considered as an entity to interface with the external environment (such as sensors). A `device` component can interact with both hardware and software components (e.g., using port connections).

Port connection A port connection allows the transfer of data and/or event between two components (`threads` and `devices`), explicitly declared between two ports. As shown in Fig. 2, the ports are directional as *in*, *out* or *in out* ports. They are also typed as data, event or event data. In the case of data transfer, the port may be typed with a `data` component.

System A `system` component is a composite of `system` components (subsystems) or of software and hardware components. Figure 2 represents an AADL model example, in which a `system` component includes three `threads` and a `device` connected through the different port connection types (data, event and event data).

3 LNT mapping for a Ravenscar task model

In this first part of our contribution, we aim to define a formal mapping for a real-time system. We use the LNT language to generically specify a standard task model. The LNT syntax combines features of programming languages with concurrency primitives adopted from process algebras, which makes it suitable for specifying concurrent tasks and handling scheduling calculations. The formal representation should simulate the scheduling, the execution and the interaction of tasks. Ideally, the task and the scheduler are separately specified to provide a comprehensible modular mapping.

In this section, we first present the considered task model. Then, we develop the proposed mapping through three principal parts: scheduling mapping, communication and composition/synchronization. Finally, we close this section with a discussion about the flexibility and the extensibility of our proposal.

3.1 A Ravenscar compliant task model

In verification context, a real-time system can be considered as a set of cooperative and concurrent tasks dispatched at regular intervals (periodic tasks) or with special events (aperiodic or sporadic tasks). The system is then abstracted as a task model with a set of real-time parameters, hiding the architectural complexity. In our work, we rely on a conventional tasking model inspired from Liu and Layland [37] real-time model. Formally speaking, we work with a set of k tasks denoted by $S = \{\tau_1, \ldots, \tau_k\}$ whose each τ_i is defined by two parameters C_i and T_i: C_i refers to the capacity or WCET (Worst Case Execution Time); and T_i is considered as the period and the relative deadline of each dispatch.

In addition, strong constraints are also considered since we deal with safety-critical systems that require certification to be used. In such a context, we mention the Ravenscar profile [13], which is defined to meet safety-critical real-time requirements (determinism, schedulability analysis, suitability for certification, etc.). This profile describes a set of restrictions of the Ada tasking features to allow the static analyses for high integrity system certifications.

The Ravenscar profile can be applied at the model level as a subset composed of a static set of tasks in interaction, run by one preemptive fixed-priority scheduler. In fact, to be Ravenscar compliant, the task model should mainly respect the following restrictions:

- All tasks must be either:
  - periodic or time-driven tasks: they are synchronous (simultaneously released at the first time) and cyclic (progress periodically) tasks;
  - sporadic or event-driven tasks: they have no fixed first activation, they are activated in response to asynchronous events (invocation-events) with a fixed minimal delay between two successive activations.
- All tasks are created at initialization and then activated and executed according to their priorities;
- All communications and synchronizations between the tasks are achieved using the protected objects[4] with these constraints:
  - at most one task can wait on each object;
  - sending and receiving operations are atomically executed through the protected object procedures.

[3] More details about our AADL subset are provided in the *AADL2LNT* transformation definition (Sect. 4).

[4] A protected object is a construction based on the well-known concept of monitors for synchronizations.

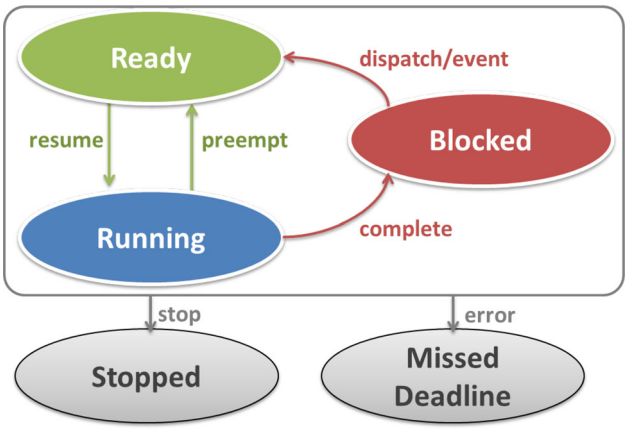

Fig. 3 Task state automaton

– Scheduling is based on FIFO-Within-Priorities policy as follows:
  – each task has a fixed priority;
  – a task may preempt a task of a lower priority.

The task execution can be represented as a state automaton drawn in Fig. 3. At any time, the task can be in one of these states (READY, RUNNING or BLOCKED). A task may be READY to run, i.e., the task is able to be executed by the processor. It can also be BLOCKED, i.e., the task cannot execute until an external event occurs. A BLOCKED task may become READY by a temporal event (dispatch for a new period) or an invocation-event (for sporadic task). The READY task can be resumed (selected by the scheduler); thus, it moves to the RUNNING state, i.e., the task is actually executing. While running, the task can be preempted and so it returns to the READY state or it can complete its execution and moves to the BLOCKED state.

Note that the task execution is simulated during a hyperperiod ($H(\tau_{1...k})$) which is the smallest time interval until the schedule repeats in a cycle of task executions. $H(\tau_{1...k})$ is calculated as the least common multiple of all task periods (LCM (T_i)), which is considered a sufficient time interval to study task models under a priority-based monoprocessor scheduling.

3.2 Scheduling mapping

The scheduling mapping concerns the task and the scheduler LNT representations. Generally described, tasks are mapped to LNT processes to be concurrently executed, each τ_i is represented by one LNT process, named `TASK`. These `TASK`s are scheduled by a main process, named `SCHEDULER`, which represents the scheduler: the `TASK`s are synchronized (through the LNT gates and channels) with the `SCHEDULER` to be activated. Note that the LNT language is not a specific real-time process algebra. It has no time operators, and all parallel processes start execution and terminate at the same time. There exist timed extensions for the CADP languages (ET-LOTOS [36], RT-LOTOS [16], etc.), but currently, they are not supported by its tools. Nevertheless, the use of the LNT language still sufficient for our purposes, since it provides a rich data part, used to specify real-time features and scheduling algorithms. Therefore, the time is a part of the proposed LNT mapping and it is smartly included (when needed) to provide LNT specifications with reduced state spaces. We define a `COUNTER` variable to represent the time (a timer to count units of time), used as required to perform the temporal calculations (e.g., dispatching, preemption). In addition, we define a `HYPERPERIOD` variable that represents $H(\tau_{1...k})$, thus, the timer `COUNTER` is bounded as ($0 <$ `COUNTER` $<$ `HYPERPERIOD`).

In the following, we, respectively, develop the `TASK` and `SCHEDULER` LNT definitions.

3.2.1 Task mapping

The `TASK` process is designed to represent the task as a schedulable concurrent unit with a potentially infinite sequence of activations (invocations or jobs) by the scheduler. To specify the considered dispatching model, we define a set of activation orders mapped to the LNT enumeration type exchanged between the `TASK` and `SCHEDULER`: `T_Dispatch_Preemption`, `T_Preemption`, `T_Preemption_Completion`, `T_Dispatch_Completion`, `T_Completion`, `T_Error` and `T_Stop`.

We include the `TASK` skeleton in Listing 9. The process declares an LNT gate, named `ACTIVATION`, to be synchronized with the `SCHEDULER`. The `TASK` behavior is an infinite `loop` whose body is a non-deterministic choice `select` in order to separate the *execution*, *error* and *termination* behaviors. The selected behavior is determined by the `ACTIVATION` communication with its different possible values.

The task state switch (Fig. 3) is mapped in the `TASK` *execution* behavior part. The `ACTIVATION` communication defines the `TASK` states: the current state is defined according to the received `SCHEDULER` order. Initially, the `TASK` is supposed at the READY state. It is suspended until the reception of a `SCHEDULER` order on the `ACTIVATION` gate. All the task transitions of Fig. 3 (`resume`, `preempt`, `dispatch` and `complete`) between the states are translated by suspensions on the `ACTIVATION` *rendezvous* with the `SCHEDULER`. At the reception of an activation order (`T_Dispatch_Completion`, `T_Preemption`, `T_Dispatch_Preemption` and `T_Preemption_Completion`), the `TASK` moves to the RUNNING

state. After the execution, the TASK sends the label T_Completion to the SCHEDULER meaning that the TASK has accomplished the activation order and it is no more at the RUNNING state. At this point, depending on the received order, the TASK may switch state as follows:

- T_Dispatch_Completion: it starts and completes the execution of the current period and enters the BLOCKED state;
- T_Dispatch_Preemption: it starts the execution in the current period but with a preemption, thus, it returns to the READY state;
- T_Preemption: it progresses in execution but without reaching the completion time, so it returns to the READY state;
- T_Preemption_Completion: it finishes the execution of the current period and enters the BLOCKED state.

The TASK can also receive a T_Error and T_Stop orders, which are, respectively, used to mark a missed deadline and to stop the system simulation. This concerns the *error* and *termination* behaviors, which leads to define two additional task states MISSED_DEADLINE and STOPPED included in the task state automaton as shown in Fig. 3, used for verification ends.

Note that periodic and sporadic tasks are represented with the same LNT skeleton, while the difference (dispatching model) will be in the activation mode controlled by the SCHEDULER. All the temporal calculations are encapsulated in the SCHEDULER which maintains that periodic tasks are executed with regular-orders, while sporadic tasks receive irregular orders according to the reception of invocation-events.

In our work, we support inter-task communications: a task can be connected with other tasks. At the LNT mapping level, exchanged data and events are generically mapped using an enumerative LNT type (labels DATA and EVENT). The connections are established through the LNT gates and channels. Thus, the TASK can have many gate declarations as required for its connections. The TASK interactions are also controlled through SCHEDULER orders that fix the input and output times as follows:

- T_Dispatch_Preemption (the start of execution time): the TASK receives inputs;
- T_Preemption_Completion (the completion time): the TASK sends outputs;
- T_Dispatch_Completion (the completion execution): the TASK receives inputs at the start time and sends outputs at the completion time.

Listing 9 TASK LNT skeleton

```
process TASK [ACTIVATION: LNT_Channel_
Dispatch,
-- other gate declarations
] is
   loop
      select
         select -- execution behavior
            -- a complete execution
 time
            ACTIVATION (T_Dispatch_
 Completion);
            []
            -- preemption
            ACTIVATION (T_Dispatch_
Preemption);
            []
            ACTIVATION (T_Preemption)
            []
            ACTIVATION (T_Preemption_
Completion);
         end select;
         ACTIVATION (T_Completion)
         []-- error behavior
         ACTIVATION (T_Error)
         []-- termination behavior
         ACTIVATION (T_Stop)
      end select
   end loop
end process
```

3.2.2 Scheduler mapping

The SCHEDULER process encodes the scheduling algorithm to simulate the execution of the tasks. It is synchronized with all the TASKs through the ACTIVATION gates. The SCHEDULER construction depends on the set of tasks S and the chosen scheduling protocol. Following the Ravenscar profile, the task execution is assumed by the preemptive fixed-priority scheduling in which priority of each task is statically fixed and the scheduler runs always the ready task with the highest priority. At any time, if a task with a higher priority becomes ready, the scheduler performs a context-switch preempting the current running task enabling the higher priority task to resume execution. In this paper, we consider the RM (Rate Monotonic) scheduling with a negligible context-switch time. The task is defined by $\tau_i = (C_i, T_i)$ whose index i represents the task priority, attributed according to its period T_i as the task with the smallest period takes the highest priority. To schedule τ_i, we also define t_j^i for the date of the jth activation and d_j^i for the date of the jth deadline.

In Listing 10, we include the SCHEDULER LNT skeleton. The process declaration has k gates (ACTIVATION_1, …, ACTIVATION_k) with n additional gates if S contains sporadic tasks (NOTIFICATION_1, …, NOTIFICATION_n, with n is the number of sporadic tasks). The SCHEDULER behavior consists of three parts as follows:

– *Initialization part*: the SCHEDULER begins with a set of initializations needed for the temporal calculations, mainly the COUNTER and the set of tasks S.
– *Operational part*: this part implements the scheduling algorithm. While COUNTER has not reached the HYPERPERIOD, the SCHEDULER simulates the execution of tasks using Algorithm 1 (illustrated in Figs. 5 and 4).
– *Stopping part*: the termination of tasks is not allowed in the Ravenscar profile, but in our context of formal verification, we define a global system termination when COUNTER = HYPERPERIOD. Therefore, the SCHEDULER sends the T_Stop order for all the tasks to mark the end of the simulation.

Listing 10 LNT SCHEDULER skeleton

```
process SCHEDULER [
   ACTIVATION_1 : LNT_Channel_Dispatch, ...,
   ACTIVATION_k : LNT_Channel_Dispatch,
   NOTIFICATION_1 : LNT_Channel_Event, ...,
   NOTIFICATION_n : LNT_Channel_Event]
is
  var    -- initialization part
     S : LNT_Type_Task_Array,
     ..
  in
     S [i] := LNT_Type_Task (..);
     ..
     loop
        if (Counter < HYPERPERIOD) then
        -- operational part
           -- time allocation
           -- update task state
           -- task activation
           -- notification for sporadic task
        else -- termination part
           ACTIVATION_1 (T_Stop) ...
           ACTIVATION_K (T_Stop)
        end if
     end loop
  end var
end process
```

The set of tasks S is statically included using an LNT array. Tasks are indexed within their fixed priorities according to their periods ($T_{1...k}$), τ_1 (index 1) has the highest priority and task τ_k (index k) the lowest one. This array represents the ready-queues in real systems: ready tasks are inserted/deleted at the head/tail according to their priorities in the ready-queue and at any time the scheduler selects the task with the highest priority for execution. In our mapping, we use a static structure where tasks have fixed indexes to mark their priorities, while their states are modifiable by the SCHEDULER itself. Each τ_i is represented with an LNT array containing a set of fixed parameters and updated states (12 elements), mainly (C_i, T_i, t^i_j, d^i_j) and it is initialized as $\tau_i = (C_i, T_i, t^i_0 = 0, d^i_0 = T_i)$.

Note that a sporadic task is ignored until the reception of an invocation-event (considered at the BLOCKED state). Thereby, the SCHEDULER should be notified for every new incoming invocation-event, which is ensured by the NOTIFICATION_i gates.

Algorithm 1: *Operational part* algorithm

```
1  begin
2    while (i ∈ S) do
3      if (Blocked or Miss_Deadline τ_i) then
4        Move to τ_{i+1};
5
6      else if (Ready τ_i) then
7        Calculate Allocated_Time of τ_i;
8        hp(i) = 1...(i − 1);
9        while (h ∈ hp(i)) do
10         if (τ_i reaches t*^h) then
             /* τ_i is preempted by τ_h        */
11           Update Allocated_Time of τ_i;
12         end if
13       end while
         /* if τ_i respects all t*^{hp(i)}, the
            Allocated_Time contains the
            required time to complete the
            execution                          */
14       Update τ_i state;
15       Activate τ_i;
16       Check sporadic tasks;
         /* if we have a preemption, we
            return to τ_h; if we have a
            notification from a sporadic
            task, we return to τ_1; else we
            move to τ_{i+1}                    */
17
18   end while
19 end
```

During the scheduling, the SCHEDULER visits the tasks in loops in order of their priorities to find and run the ready tasks. From the highest to the lowest priority, each task is handled to determine its current state (READY or BLOCKED), thus the first task τ_i fixed to the READY state, is always the ready task with the highest priority as shown in Fig. 4. Formally, the SCHEDULER compares t^i_j and d^i_j with the COUNTER value. Thus, it decides τ_i state as follows:

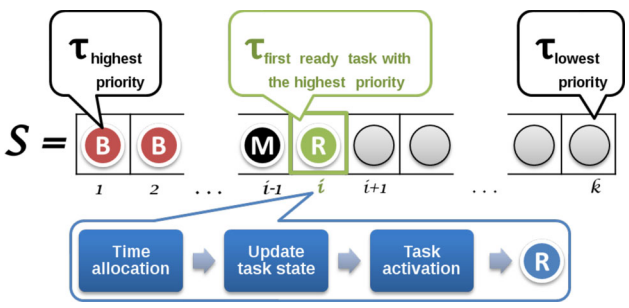

Fig. 4 SCHEDULER algorithm: ready task

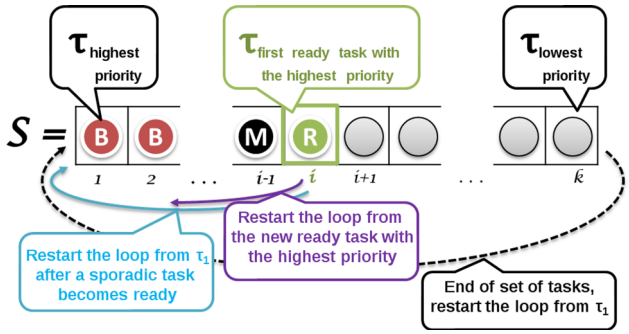

Fig. 5 SCHEDULER algorithm: task-loops

- BLOCKED: τ_i is an inactive sporadic task or it is a periodic task awaiting for the next dispatch (t^i_j >COUNTER);
- MISSED_DEADLINE: τ_i has missed a deadline (d^i_j < COUNTER), in this case, the T_Error label will be sent to the task;
- READY: τ_i is initialized, preempted or dispatched (t^i_j ≤COUNTER< d^i_j).

The BLOCKED or MISSED_DEADLINE tasks are ignored and the SCHEDULER moves to τ_{i+1}. Else, if τ_i is asserted at the READY state, the SCHEDULER decides about the execution of τ_i which moves to the RUNNING state.

At that level, the SCHEDULER can execute τ_i with its whole capacity C_i, and then, it moves to handle other tasks. This can be sufficient for a non-preemptive scheduler of a set of periodic tasks. However, in our context, a task with a higher priority can become ready at any time. Similarly, a sporadic task with a higher priority can become ready by the reception of an invocation-event. These cases can lead to preempt the execution of the current running task, so they should be considered during the scheduling. The idea consists in interrupting the SCHEDULER task-loops and restarting the task-loop to consider new ready tasks as shown in Fig. 5. In the remainder of this section, we explain this algorithm through three steps achieving the execution of a given READY task τ_i: *time allocation*, *update task state* and *task activation*. In addition, we include a *sporadic task checking* section added if S contains sporadic tasks.

Time allocation This step calculates the execution time of the current period for τ_i and prepares an activation order that will be sent at the *task activation* step. We remind that τ_i is the current ready task with the highest priority. We define the variable Allocated_Time to compute the execution time. It can return the required time to achieve the execution in the current period, or it can return a part of the execution time, since τ_i can be preempted by another task ($\tau_{h<i}$) which is a new ready task with a higher priority. For this reason, in the calculation of the Allocated_Time value, SCHEDULER should always check the states of tasks with higher priorities. We define $hp(i) = \{1 \dots i-1\}$ the set of indexes of tasks with higher priorities than τ_i. Simply, the SCHEDULER checks $t^{hp(i)}_{j+1}$ which are the next activation times of $\tau_{hp(i)}$ by comparing COUNTER + Allocated_Time value with all $t^{hp(i)}_{j+1}$ value as shown in Algorithm 1 (lines 7–12). Three alternatives can be presented leading to fix different activation orders:

1. a complete execution time C_i is allocated, if τ_i respects all $t^{hp(i)}_{j+1}$ (COUNTER + Allocated_Time < $t^{hp(i)}_{j+1}$): T_Completion_Execution order
2. a preemption is imposed, if Allocated_Time+ COUNTER reaches an $t^{h<i}_{j+1}$ of $\tau_{h<i}$ (with $h \in hp(i)$), thus (Allocated_Time=$t^{h<i}_{j+1}$-COUNTER) with two behaviors:
   2.1. a preemption at the start time of the execution: T_Dispatch_Preemption order
   2.2. a preemption in the middle of the execution: T_Preemption order
3. a complete needed time is allocated when τ_i is already preempted: T_Preemption_Completion order

Note that in the case of a preemption (alternative 2), τ_i is preempted by $\tau_{h<i}$ so the SCHEDULER restarts its task-loop from h to handle $\tau_{h<i}$, the new ready task with the highest priority as shown in Fig. 5.

Update task state At this point, τ_i is considered at the RUNNING state, the SCHEDULER increments the COUNTER with the Allocated_Time and updates the task array for the next activations. In the case of a preemption, the SCHEDULER conserves the task state and saves the executed time of τ_i in order to complete the rest later. In the case of a non-preemption, the SCHEDULER prepares the task for a new period. A periodic task becomes (C_i, T_i, $t^i_{j+1} = d^i_j$, $d^i_{j+1} = d^i_j + T_i$). In contrast, the parameters of a sporadic task cannot be predicted. the SCHEDULER has no values for its next activation or deadline. Currently, the sporadic task is viewed as (C_i, T_i, $t^i_{j+1} \geq d^i_j$, $d^i_{j+1} = \infty$) and it is ignored in the scheduling until the reception of a new notification.

Task activation The SCHEDULER sends to τ_i its current order with the ACTIVATION_i gate. It waits for a T_Completion notification from τ_i and then moves to τ_{i+1} calculation. In the case of a missed deadline, the T_Error label will be the last order sent to τ_i, since it will be ignored in the rest of the simulation.

Sporadic task checking The global state of the set of tasks may change after each task activation (exchanging events

and data, increase of the COUNTER, etc.). Particularly, the sporadic tasks may be activated by the invocation-events and may move to the READY state, so they should be considered in the scheduling with other periodic tasks. To this end, after each task activation, the SCHEDULER consults all the NOTIFICATION_i gates. When receiving a notification at $t_a'^i$, the SCHEDULER applies the following steps:

1. marking $t_a'^i$ equal to the current value of the COUNTER;
2. verifying $t_a'^i \geq t_{a-1}^i + T_i$: a notification can be ignored. Since we consider T_i as the minimal delay between two successive activations, τ_i cannot be reactivated before $t_a^i + T_i$ ($t_{a+1}'^i \geq t_a^i + T_i$);
3. if (2) is verified, then, τ_i moves to the READY state with these parameters (C_i, T_i, $t_a^i = t_a'^i$, $d_a^i = t_a^i + T_i$). Thus, τ_i is considered in the scheduling and served according to its priority;
4. if (2) is verified, then the SCHEDULER restarts the task-loop (return to τ_1): the SCHEDULER should recheck the set of tasks to find the new ready task with the highest priority, as shown in Fig. 5.

3.3 Communication

In our work, the tasks can be connected to each other to asynchronously exchange data or events. In the sporadic case, each task has at least one connection needed for its activation: the reception of an invocation-event activates the sporadic task that may move to the READY state.

The LNT processes communicate by bidirectional blocking *rendezvous* on gates. The LNT *rendezvous* on a gate allows the synchronization of n processes (several sending and receiving offers at the same time). In our case, we do not need such an advanced synchronization between processes. We consider the gates unidirectionally and we use only the synchronization between pair of processes (sender and receiver). The asynchronous inter-task connections cannot be mapped directly with synchronizations of the LNT gates since they denote blocking *rendezvous*. For this reason, we add an auxiliary process the CONNECTOR to represent the connection by means of the Ravenscar protected objects. CONNECTOR has two main gates (INPUT and OUTPUT) and a variable to save data/event.

In Listing 11, we give the CONNECTOR skeleton. The process behavior consists of three parts through an infinite `loop` whose body is a `select` statement to separate the *sending* and *receiving* data/event and the *sporadic notification*. Thus, only one operation can be executed at any time and the choice is solved by the possibility of communication on the gates.

Listing 11 LNT CONNECTOR skeleton

```
process CONNECTOR [
   INPUT: LNT_Channel_Port,
   OUTPUT: LNT_Channel_Port,
   NOTIFICATION: LNT_Channel_Event]
   (Queue_Size: Nat)
is
   loop
      select
         -- inputs of event/data part
         INPUT  ()
         []
         -- output of event/data part
         OUTPUT ()
         []
         -- sporadic part
         -- needed to notify SCHEDULER
         -- when receiving a new event
         NOTIFICATION ()
      end select
   end loop
end process
```

Each connection between two TASKs is mapped to a CONNECTOR synchronized (*rendezvous* point) with the sender on INPUT and the receiver on OUTPUT which assumes the atomicity of the two operations (sending and receiving) and the TASK unicity in awaiting at any time (respectively, on the INPUT and the OUTPUT gates).

Data are saved and kept until the next reception. Each time a new input is received, the last one is overwritten. In contrast, events are queued in an LNT list with a defined size. We use non-blocking FIFO, in which the new incoming input overwrites the previous events in the case of an overflow. In the case of an empty FIFO, the TASK receives an EMPTY message without blocking.

Since we consider a special invocation-event for the sporadic task activation, we add a third gate to the CONNECTOR process (named NOTIFICATION) to be synchronized with the SCHEDULER. We use this gate to notify the SCHEDULER of every new reception; thus, it considers the concerned task in the scheduling.

3.4 Composition and synchronization

We complete the proposed LNT mapping with two mandatory steps: composition and synchronization. All described LNT processes should be structured (connected) to form the main system. This step is ensured by the LNT `par` composition statement for the parallelism and the synchronization (global and interface) of the TASK, CONNECTOR and SCHEDULER processes. Thus, we assemble the whole system within the MAIN process.

Note that a set of the LNT types and channels are used for the TASK-CONNECTOR, CONNECTOR-SCHEDULER and TASK-SCHEDULER synchronizations. For example, we include in Listing 12 the type and channel for the TASK-SCHEDULER synchronization.

Listing 12 TASK-SCHEDULER: LNT type and channel

```
type LNT_Type_Dispatch is
   T_Dispatch_Completion,
   T_Dispatch_Preemption,
   T_Preemption,
   T_Preemption_Completion,
   T_Completion,
   T_Error,
   T_Stop
end type

channel LNT_Channel_Dispatch is
   (LNT_Type_Dispatch)
end channel
```

For further explanations, we include an example of a task model whose MAIN process is included in Listing 13 and graphically presented in Fig. 6. The initial task model (τ_1, τ_2) consists of a periodic task connected to another sporadic task running on a scheduler (*Producer–Consumer* system). The obtained LNT specification contains five processes synchronized in the MAIN process. The par composition is globally used for the following synchronizations:

- The TASK_CONSUMER and CONNECTOR are synchronized on the RECEIVE_A gate;
- The CONNECTOR and TASK_PRODUCER are synchronized on the SEND_A gate;
- The CONNECTOR and SCHEDULER are synchronized on the NOTIFICATION_1 gate;
- The TASK_CONSUMER, TASK_PRODUCER and SCHEDULER are synchronized on the ACTIVATION_1 and ACTIVATION_2 gates.

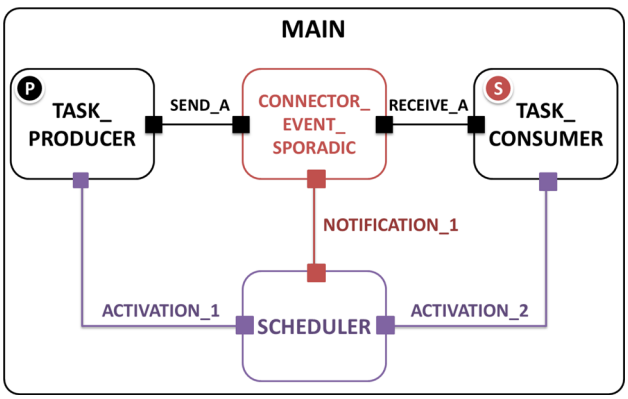

Fig. 6 *Producer–consumer*: LNT graphical MAIN

Listing 13 *Producer–consumer*: LNT code MAIN

```
process MAIN [
   ACTIVATION_1: LNT_Channel_Dispatch,
   ACTIVATION_2: LNT_Channel_Dispatch,
   NOTIFICATION_1: LNT_Channel_Event,
   SEND_A: LNT_Channel_Port,
   RECEIVE_A: LNT_Channel_Port
] is
   par
    ACTIVATION_1, RECEIVE_A ->
    TASK_CONSUMER[ACTIVATION_1, RECEIVE_A]
    ||
    NOTIFICATION_1, SEND_A, RECEIVE_A ->
    CONNECTOR[SEND_A, RECEIVE_A, NOTIFICATION_1]
    ||
    SEND_A, ACTIVATION_2 ->
    TASK_PRODUCER[ACTIVATION_2, SEND_A]
    ||
    ACTIVATION_1, ACTIVATION_2, NOTIFICATION_1 ->
    SCHEDULER[ACTIVATION_1, ACTIVATION_2,
              NOTIFICATION_1]
   end par
end process
```

3.5 Discussion

The obtained LNT specification (the set of the LNT definitions composed in the MAIN process) represents a formal executable semantics for a real-time task model, where the TASKs are connected through the CONNECTORs and scheduled by the SCHEDULER. This mapping is flexible enough to support various task models with minor changes: periodic/sporadic tasks, independent/communicating tasks and preemptive/non-preemptive tasks.

In addition, the real-time features are modularly designed which increases the extensibility of the mapping: each feature can be separately completed or extended. For example, the TASK can be easily enriched with more behavior that can be specified in a separate LNT process or function and just called within the TASK skeleton.

Similarly, the scheduling mapping can be extended with other scheduling protocols since we specify an explicit SCHEDULER that encapsulates all the scheduling calculations. For example, we have developed the EDF (Earliest Deadline First) based on unfixed priority scheduling. The SCHEDULER skeleton is conserved. The modifications can be limited in the *operational part*, mainly, the *time allocation* operation. The other manipulations (*update task state*, *task activation* and *sporadic task checking*) can be conserved, and thus the TASK and CONNECTOR processes need no changes.

4 Model transformation

In this section, we discuss the applicability of our LNT mapping as a software engineering practice. We aim at integrating the formal verification in a model-driven process based on

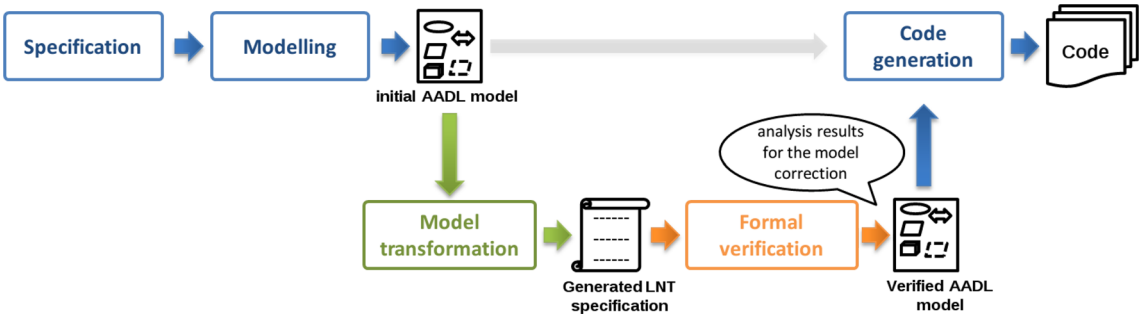

Fig. 7 AADL model-based development process

the AADL language. A basic proposal was discussed in [43], where we presented a transformation from AADL into LNT about the periodic `threads` with a preemptive RM scheduler. In this paper, we propose to refine this approach and extend the scope of our work.

As simply depicted in Fig. 7, throughout the development process, the system representation takes different forms (written specification, architectural model, source code) on many phases (specification, modeling and code generation). During the modeling phase, the formal verification of the AADL model seems useful and complementary to traditional syntactic and semantic analyses. To this end, we define the *AADL2LNT* model transformation from the initial AADL model into an LNT specification based on the proposed LNT mapping. This transformation achieves the automatic generation of an LNT specification compliant with the CADP toolbox and ready for verification.

In the remainder of this section, we develop our AADL subset. Then, we describe the *AADL2LNT* transformation rules with a discussion about the novelty and the improvements compared to our previous work [43]. Finally, we describe our implementations allowing the definition of a tool-chain for a development process based on the AADL language.

4.1 AADL subset

The AADL language describes different concepts of real-time embedded systems with a rich semantics detailed in its standard [1]. The language covers many important aspects (timing requirements, fault and error behaviors, time and space partitioning, safety properties, etc.) that cannot be wholly analyzed in one approach.

According to our verification purposes, we define an AADL subset whose elements are depicted in Fig. 8. Moreover, the consideration of the Ravenscar profile requires some additional restrictions applied at the model level, meaning that the AADL subset should be compliant with the profile. Mainly, the profile claims that the `threads` are either sporadic or periodic and they are schedulable according to the Rate Monotonic analyses such as the *exact schedulability test* [37]: for a synchronous set of tasks S of n independent and periodic tasks, if $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right) \approx 0.69$, then S is RM-schedulable.

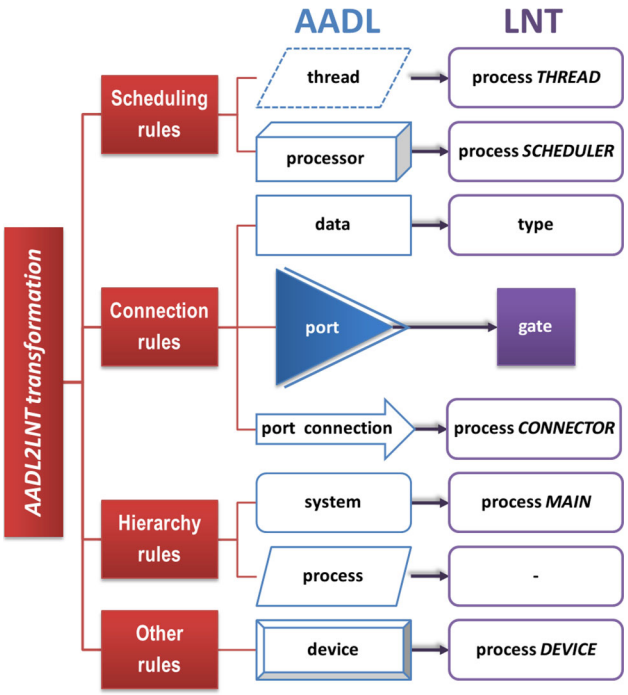

Fig. 8 High-level view of *AADL2LNT* transformation

In our work, we aim at defining an executable formal semantics of the AADL model viewed as a set of communicating tasks in a real-time context. This abstraction allows different alternatives of verification, such as the schedulability analysis, the `thread` execution simulation and the verification of communication properties.

At the model level, we consider the `system` as a set of `threads` bound to a monoprocessor and communicating through the port connections . We do not support either AADL shared access or AADL flows/modes. The model is completed by a set of standard properties

attributed to different components. We support the following lists:

- properties specifying the constraints for the software-hardware binding like the Actual_Processor_Binding property, which can list only one processor;
- properties specifying the temporal and scheduling information such as Dispatch_Protocol (periodic or sporadic), Compute_Execution_Time, Period, Input_ Time (Start_Time for all thread ports) and Output_ Time (Completion_Time for all thread ports);
- properties specifying the information for the port connections like Queue_Processing_Protocol (FIFO), Queue_Size, Overflow_Handling_Protocol (drop oldest) and Dequeue_Protocol (one item).

In the AADL language, the system is modeled through a set of nested components whose top-level is the system component, as shown in Fig. 2 and Table 3.

The system implementation mainly contains: a set of subcomponents that can be data, process, processor or device; a set of connections to declare port connections of processes and devices; and a set of properties in which the Actual_Processor_Binding property is used to bind the processor with the processes.

The *AADL2LNT* transformation can be applied on an *instantiable* AADL system, which means that the model is successfully analyzed (syntactically and semantically) and can be completely bound, as when all threads are bound to the processor.

4.2 Transformation rules

Model transformation plays a crucial role in MDE for various goals (modeling, optimization and analysis). It is the mechanism of generating a target model based on information extracted from a source one. This operation is based on our LNT mapping and requires a set of new LNT definitions to cover the considered AADL subset. The *AADL2LNT* transformation is described with a set of correspondence rules between AADL and LNT (summarized in Tables 1, 2, 3 and 4).

Basically, the LNT mapping (Sect. 3) is used as follows: we translate each AADL thread component into a TASK process; we represent the AADL ports by the LNT gates; the AADL port connections are mapped through the CONNECTOR processes; and the AADL processor becomes the SCHEDULER process. Figure 8 graphically represents these basic transformation rules, which are applied through four steps, respectively, developed in the rest of this section.

4.2.1 Scheduling rules

The execution and scheduling rules concern the thread and processor components.

Thread rule The AADL thread is a schedulable unit that can be concurrently executed with other threads. Each thread executes a set of sequential instructions. It is declared within a process component, and it is bound to a processor component to be scheduled.

The thread component becomes the TASK process described in Sect. 3.2.1. As illustrated in Table 1, the TASK process takes the thread implementation name prefixed by "THREAD_" and declares the ACTIVATION default gate to be synchronized with the SCHEDULER process.

The supported standard properties are used to specify the temporal parameters of the thread, as follows: the Dispatch_Protocol property represents its dispatch model; the Period property represents its period T_i; and the Compute_Execution_Time property represents its capacity C_i.

In the AADL language, the dispatch semantics is given for the standard dispatch protocols (periodic, sporadic, aperiodic, timed, hybrid, and background threads). Since we consider a Ravenscar compliant model, we support periodic and sporadic dispatch models, as described in the AADL standard:

- periodic threads: they are periodically dispatched at time intervals of the specified Period property value.
- sporadic threads: they are activated as the result of an event/event data (invocation-event) arriving at an event/event data port of the thread. The time interval between two successive dispatch requests will never be less than the associated Period property value.

As described in Sect. 3.2.1, the thread dispatching is ensured by the SCHEDULER through the defined activation orders (T_Completion, T_Dispatch_Preemption, T_Preemption_Completion, T_Dispatch_Completion and T_Preemption).

The dynamic semantics for an AADL thread is defined using a hybrid automaton (*thread scheduling and execution states* automaton) included in Fig. 9. The TASK mapping covers an important part of the AADL thread semantics of its dispatching, its scheduling and its execution. Compared to the standard thread hybrid automaton, the proposed state automaton of Fig. 3 excepts the awaiting states (shared resources, subprogram calls and background thread), that are non-covered by our work since we do not support either subprogram components or shared resources. In addition, the standard defines a sus-

Table 1 Scheduling rules

AADL	LNT
AADL `thread` transformation rule	

`thread T`
`  properties`
`  Dispatch_Protocol => Sporadic/Periodic;`
`  Compute_Execution_Time => min .. $C_i$;`
`  Period => $T_i$;`
`end T;`
 	
`thread implementation T.Impl`
 	
`end T.Impl;`
	`process THREAD_T_IMPL [`
	`  ACTIVATION:LNT_Channel_Dispatch]`
	`is`
	`  loop select`
	`  select`
	`  ACTIVATION (T_Dispatch_Preemption)`
	`  []`
	`  ACTIVATION (T_Preemption_Completion)`
	`  []`
	`  ACTIVATION (T_Dispatch_Completion)`
	`  []`
	`  ACTIVATION (T_Preemption)`
	`  end select;`
	`  ACTIVATION (T_Complete)`
	`  []`
	`  ACTIVATION (T_Error)`
	`  []`
	`  ACTIVATION (T_Stop)`
	`  end select end loop`
	`end process`

AADL `processor` transformation rule	

`processor the_processor`
`  properties`
`  Scheduling_Protocol => RMS;`
`end the_processor;`
	`process SCHEDULER [`
	`  ACTIVATION_1: LNT_Channel_Dispatch, ...,`
	`  ACTIVATION_K: LNT_Channel_Dispatch,`
	`  INCOMING_EVENT_1: LNT_Channel_Event, ...,`
	`  INCOMING_EVENT_N: LNT_Channel_Event]`

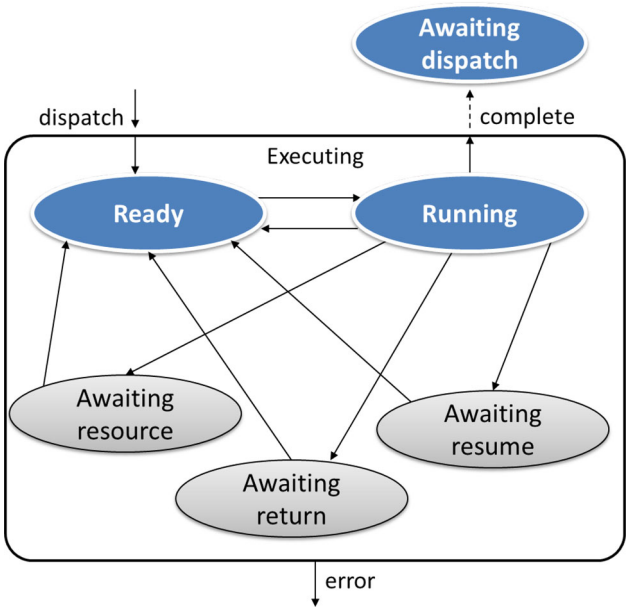

Fig. 9 AADL `thread` hybrid automaton [1]

pended AWAITING DISPATCH state for `threads` when completing the execution of the current dispatch, which corresponds to the BLOCKED state in the proposed LNT mapping.

Processor rule The `processor` is a hardware component that ensures the scheduling and execution of the `threads`. As shown in Table 1, the `processor` component becomes the `SCHEDULER` process developed in Sect. 3.2.2. Every link between the `processor` and a `thread` corresponds to an `ACTIVATION_i` gate declaration in the `SCHEDULER` process. In the case of a sporadic `thread`, the corresponding `NOTIFICATION_i` gate is also declared.

The task model $S = \{\tau_1, \ldots, \tau_k\}$ with $\tau_i = (C_i, T_i)$, is extracted from the AADL model for the `SCHEDULER` generation. We distinguish different information required for the scheduling: the number of `thread` instances (k); the number of sporadic `thread` instances n; the set of values of each `thread` properties ($T_{1\ldots k}$, $C_{1\ldots k}$) to compute the hyperperiod $H(\tau_{1\ldots k})$, assign each `thread` priority and encode the task LNT array `S` of the *initialization part* (Sect. 3.2.2, Listing 10).

4.2.2 Connection rules

The `thread` components may declare (data, event or event data/*in*, *out* or *in out*) ports to be in interaction with other components. A port connection allows the transfer of data and/or event between two components, explicitly declared between two ports at `process` and `system` levels. The connection rules concern then the AADL ports (typed with

Table 2 Connection rules

AADL	LNT
AADL `data` transformation rule	

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

AADL port transformation rule	

`thread TH`
`  features`
`  A : in event data port T;`
`  properties`
`  Dispatch_Protocol => Sporadic/Periodic;`
`  Compute_Execution_Time => min .. $C_i$;`
`  Period => $T_i$;`
`end TH;`
	`process Thread_TH [ACTIVATION: LNT_Channel_Dispatch,`
	`  PORT_A: LNT_Channel_Port]is`
	`  var  A : LNT_Type_Data in`
	`  A := EMPTY;`
	`  ...`
	`  select`
	`  ACTIVATION (T_Dispatch_Completion);`
	`  PORT_A (?A);`
	`  []`
	`  ACTIVATION (T_Dispatch_Preemption);`
	`  PORT_A (?A);`
	`  [] ...`
	`  end select;`
	`  ACTIVATION (T_Completion)`
	`  ...`
	`end process`

AADL port connection transformation rule	

`process[system] implementation P.Impl`
`subcomponents`
`  A: thread[process] AA; B: thread[process] BB;`
`connections -- data port`
`  AB: port A.a1 -> B.b1;`
	`process PERIODIC_DATA_CONNECTOR`
	`[INPUT: LNT_Channel_Port,`
	`  OUTPUT: LNT_Channel_Port]`

`process[system] implementation P.Impl`
`subcomponents`
`  A: thread[process] AA; B: thread[process] BB;`
`connections -- event [event data] port`
`  AB: port A.a1 -> B.b1;`
	`process PERIODIC_EVENT_CONNECTOR`
	`[INPUT: LNT_Channel_Port,`
	`  OUTPUT: LNT_Channel_Port]`
	`(Queue_Size:Nat)`

`process[system] implementation P.Impl`
`subcomponents`
`  -- sporadic thread`
`  A: thread[process] AA; B: thread[process] BB;`
`connections -- event [event data] port`
`  AB: port A.a1 -> B.b1;`
	`process SPORADIC_EVENT_CONNECTOR`
	`[INPUT: LNT_Channel_Port,`
	`  OUTPUT: LNT_Channel_Port,`
	`  NOTIFICATION : LNT_Channel_Event]`
	`(Queue_Size:Nat)`

the `data` components) and their connections, which are declared at the `process` and `system` levels.

Data rule The `data` component is mapped into a generic LNT type. For a full abstraction, we use a generic type for all data and event exchanging (label `AADLDATA` as included in Table 2). A corresponding channel is also added to allow the communication. At this level, this representation is sufficient to draw connections between the `threads` since we do not handle the AADL `data` content.

Port rule The AADL ports are declared in the `thread` component for the transfer of control and data. They are transformed into the LNT gate declarations. Since they are bidirectional, the LNT gates can represent *in* and *out* ports. In addition, we complete the behavior of the `TASK` skeleton of Listing 9 with an *initialization* part using the LNT `var` statement. As shown in Table 2, for each declared port, named `A`, we proceed as follows:

- a gate declaration `PORT_A:LNT_Channel_Port` is added;
- a variable `A:LNT_Type_Data` is declared and initialized in the *initialization* part;
- a corresponding communication is added as follows:
  - for *out* port: `PORT_A (!A)`
  - for *in* port: `PORT_A (?A)`

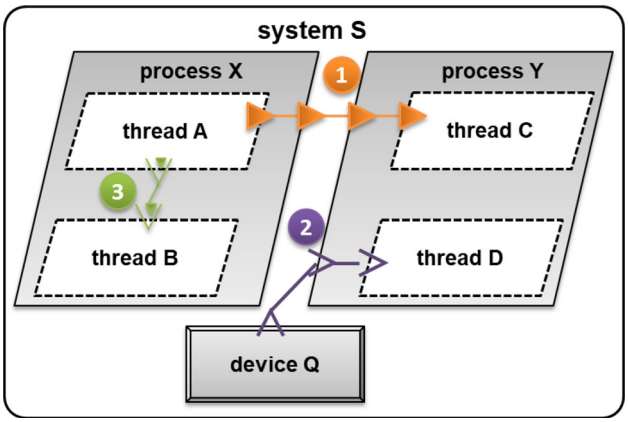

Fig. 10 AADL semantics port connections

Note that data, event or event data ports are exactly mapped at the THREAD_* level, while the difference between these types (reception, queuing, etc.) is assumed by the communication mechanism using the CONNECTORs.

Port connection rule The port connections are ensured through the synchronizations of the CONNECTOR instances (Sect. 3.3) allowing unidirectional communications, so we consider only 1-to-1 connections with no *in out* ports. Each port connection becomes a CONNECTOR instance. The CONNECTOR should be synchronized on the INPUT and OUTPUT gates between two TASKs equivalent, respectively, to the threads of *in* and *out* ports.

In the AADL language, the port connection declarations follow the containment hierarchy of the threads, processes and systems or of devices and systems, as shown in Fig. 10. The CONNECTOR process represents the semantics port connection abstracting all the port connection declarations that follow the component containment in the instantiated system from an ultimate source (*out* port of a thread or a device) to an ultimate destination (*in* port of a thread or a device). Thus, the connection ❶, ❷ or ❸ of Fig. 10 is similarly transformed into a CONNECTOR instance, despite the difference in the declaration-level.

In our work, we consider the AADL asynchronous connections whose determinism is ensured by the Ravenscar constrained protected object as developed in Sect. 3.3. In the AADL semantics port connection, the content of incomings is frozen during the thread execution: the port variable content is not affected by the arrival of new incomings. Data and events arriving through *in* ports are available to the thread at a specified input time, fixed by the Input_Time property. This communication model is assumed in the LNT specification through the INPUT synchronization (reading the port content) between the CONNECTOR and the THREAD_* corresponding to the thread of *in* port. According to our LNT mapping, the INPUT *rendezvous* is fixed at the start time of each period equivalent to the Start_Time value of the Input_Time property. After the INPUT *rendezvous*, any new data or event arriving becomes available only at the next start time. In addition, the AADL port output is transferred to the other components at an output time specified by the Output_Time property. The transfer of data or event corresponds to the OUTPUT synchronization with the thread of *out* port, which is fixed at the completion time of each period equivalent to the Completion_Time value of the Output_Time property.

According to the port type (data, event or event data) and the thread Dispatch_Protocol property value (periodic or sporadic), we generate one of three CONNECTOR types, as included in Table 2. In the case of a data port and a periodic thread, the data port connection is mapped by a simple CONNECTOR without queuing or sporadic notifications. When exchanging events, we use a CONNECTOR with an input list for the event queuing, that implements the supported set of AADL properties. Finally, the NOTIFICATION gate is added in the case of a sporadic thread.

4.2.3 Hierarchy rules

The AADL components are hierarchically structured in both the process (a set of thread subcomponents) and the system components.

Process rule The process component represents a protected virtual address space that can be ignored in verification. So the processes have no equivalent in the obtained LNT specification and its dispatch semantics is omitted. The AADL model may contain a process with a composition of the threads. In this case, the corresponding THREAD_* instances are directly added in the MAIN process.

System rule The system component becomes the LNT MAIN process. This generation corresponds on the LNT synchronization and composition phase developed in Sect. 3.4. The MAIN generation can be summarized in three steps:

- Preparation of the list of thread instances : each process subcomponent corresponds to one or more of THREAD_* instances.
- *_CONNECTOR synchronizations: for each port connection, we create one *_CONNECTOR instance. We use the AADL connection name, prefixed by "SEND_" and "RECEIVE_" to represent two gates (e.g., the port connection Connect_AB, of the example of Table 3, is represented by the SEND_AB and RECEIVE_AB gates). These gates are used in the synchronization between the THREAD_* and *_CONNECTOR instances as follows:
  - the RECEIVE_* synchronization represents the reading of the port content by the thread of *in* port.

Table 3 Hierarchy rules

AADL	LNT
AADL `system` transformation rule	

`system PC`
`end PC;`
 	
`system implementation PC.Impl`
`  subcomponents`
`  Data_Exp: data Alpha;`
`  A: process AA.Impl;`
`  B: process BB.Impl;`
`  CPU: processor the_processor;`
`  connections`
`  AB:port A.A1 -> B.B1;`
`  properties`
`  Actual_Processor_Binding => (reference (CPU))`
`  applies to A;`
`  Actual_Processor_Binding => (reference (CPU))`
`  applies to B;`
`end PC.Impl;`
	`process Main [`
	`ACTIVATION_1: LNT_Channel_Dispatch,`
	`ACTIVATION_2: LNT_Channel_Dispatch,`
	`SEND_AB: LNT_Channel_Port,`
	`RECEIVE_AB: LNT_Channel_Port,`
	`INCOMING_EVENT_2: LNT_Channel_Event] is`
	`par`
	`  ACTIVATION_1,SEND_AB ->`
	`  THREAD_P_IMPL[ACTIVATION_1,SEND_AB]`
	`  ||`
	`  SEND_AB,RECEIVE_AB,INCOMING_EVENT_2->`
	`  Event_Port[SEND_AB,RECEIVE_AB,INCOMING_EVENT_2](3)`
	`  ||`
	`  ACTIVATION_2,RECEIVE_AB ->`
	`  THREAD_C_IMPL[ACTIVATION_2,RECEIVE_AB]`
	`  ||`
	`  ACTIVATION_1,ACTIVATION_2,INCOMING_EVENT_2->`
	`  SCHEDULER`
	`  [ACTIVATION_1,ACTIVATION_2,INCOMING_EVENT_2]`
	`end par`
	`end process`

Table 4 Other rules

AADL	LNT
AADL `device` transformation rule	

`device D`
`  features`
`  A: out event port;`
`  B: in event port;`
`end D;`
	`process Device_D [PORT_A: LNT_Channel_Port,`
	`  PORT_B: LNT_Channel_Port]`
	`is`
	`  var A: LNT_Type_Data, B: LNT_Type_Data in`
	`  A := AADLDATA; B := EMPTY;`
	`  loop select`
	`  PORT_A (?A)`
	`  []`
	`  PORT_B (!B)`
	`  end select end loop`
	`  end var`
	`end process`

  - the `SEND_*` synchronization represents the transfer of the data or event from the `thread` of *out* port.
- Global composition: all `THREAD_*`s are synchronized with the `SCHEDULER` on `ACTIVATION_i` gates. Similarly, all `*_CONNECTOR`s are synchronized with the `SCHEDULER` on the `NOTIFICATION_i` gates.

For the `system` transformation illustration, this rule is applied on an AADL simple *Producer–Consumer* model (two AADL `threads` running on one `processor` to exchange events), which is included in Table 3.

4.2.4 Other rules

At this level, the LNT mapping is used in the definition of the *AADL2LNT* transformation with minor changes. In addition, the transformation can be completed by rules concerning other AADL components such as `devices`.

Device rule The `thread` components can communicate with `devices` via ports. We do not consider the internal `device` behavior, but we provide a simple LNT specification sufficient for the `thread-device` port connections.

The `device` becomes an LNT process prefixed by "`DEVICE_`". Unlike `THREAD_*`, `DEVICE_*` has no activation gate, but its port declarations are similarly mapped. The `DEVICE_*` behavior consists simply of a loop-select statement comprising the `PORT_*` communications as given in Table 4. The AADL `device` port connections are similarly mapped as the `thread` port connections through the `CONNECTOR`s at the `MAIN` level.

4.3 Discussion

The description of a model transformation based on more generic LNT definitions was the purpose of our previous work published in [43] and depicted in Fig. 11 (the *Producer–*

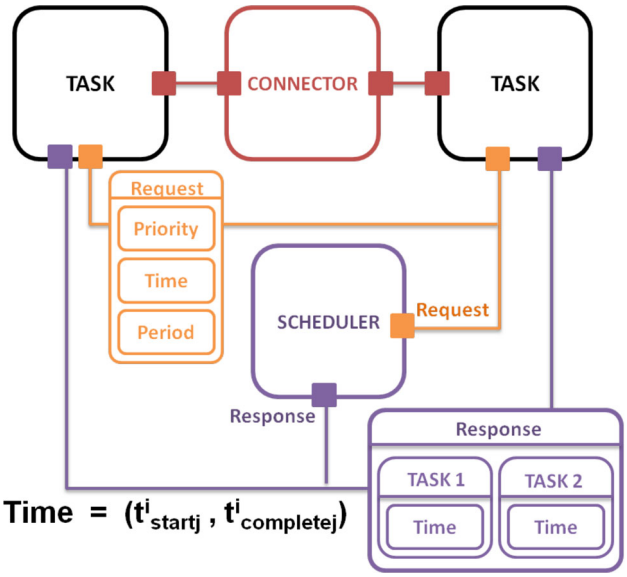

Fig. 11 Old LNT mapping

consumer example). Mainly, the scheduling was differently ensured between the THREAD_*s and SCHEDULER. Compared to the actual SCHEDULER, the old one was generically defined to execute any task model: it has no information about the task model (no *initialization* part). Using only two gates (REQUEST and RESPONSE), the SCHEDULER gives a response according to a received request without a global view of the task model: the execution advances by exchanging time $[t^i_{start_j}, t^i_{complete_j}]$ between the SCHEDULER and THREAD_*s. Each process (THREAD_* and SCHEDULER) has a local COUNTER. Each THREAD_* requests time for its execution $[(t^i_{start_j}, t^i_{complete_j}), T_i, i]$ and waits for the SCHEDULER response. While the SCHEDULER computes start and completion execution time based on the tasks priorities and the advancement of its COUNTER. The THREAD_* manages its own temporal calculations (execution times, preemptions, etc.). At the reception of a response, the THREAD_* updates its COUNTER, executes the allocated time and then prepares a new request.

This first proposition brings generic SCHEDULER and THREAD_* constructions, favoring the reuse of our mapping. However, in the case of large systems, this mapping rapidly led to the state explosion problem during the analysis phase: the system state space may become very large, or even infinite, that cannot be explored with limited resources of time and memory (see Sect. 5.5). In this paper, we redefine the SCHEDULER and THREAD_* processes and their synchronizations in order to avoid and reduce this problem. We opt to get rid of the heavy requests and responses by abstracting the SCHEDULER-THREAD_* exchanges with a set of activation orders to restrict the enumerations in the analysis phase. In addition, we eliminate the time counters from the THREAD_*s and maintain a unique counter of the whole system within the SCHEDULER. Thus, useless calculations are removed and time is managed only by the SCHEDULER.

For the same reasons, some restrictions were applied on the considered AADL subset concerning mainly the port connection rules. The *in out* ports and n-to-n connections can be supported in the *AADL2LNT* transformation since the LNT language provides bidirectional gates and n-to-n synchronizations. However, the resulting formal specifications rapidly explode, especially with highly connected models.

With these refinements, the resulting state spaces are significantly reduced at the analysis phase. Thus, we provide a scalable solution (see evaluation in Sect. 5.5) without restricting our purposes (preemptive priority-based scheduling, asynchronous communication, etc.).

In another direction, we aim to enrich the communication mapping by considering the content of exchanges. The *AADL2LNT* transformation is completed by the consideration of the AADL Behavior annex [2], which is used to specify the behavior handling inputs and outputs within the thread components. The mapping of the Behavior annex requires a new abstraction level where the thread behavior (described as a local state transition machine) and data content are considered in a new AADL subset. The Behavior annex transformation with the set of new rules about AADL thread and data components can be found in [42].

4.4 Tool-chain

The model transformation description being defined, we reach the implementation phase to provide an automatic generation of the LNT specification from a given AADL model. A detailed description of our implementations can be found in [44]. In this section, we briefly describe the obtained tool-chain, depicted in Fig. 12, based on Ocarina [35] for architectural modeling and CADP [22] for formal verification:

- Ocarina[5] is an open-source tool suite for AADL modeling developed since 2004 and deployed on GitHub under the OpenAADL project. Ocarina can be used as a standalone compiler for the AADL language with the support of some annexes ARINC653, EMV2 and REAL. The tool suite is appropriate for an MDE approach since it provides basic analysis (syntactic and semantic), advanced model manipulations, formal verification (Petri nets) and code generation (toward the AADL runtime PolyORB-HI/Ada and C).
- CADP is a toolbox for the design and verification of concurrent systems, developed since 1986. It is avail-

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

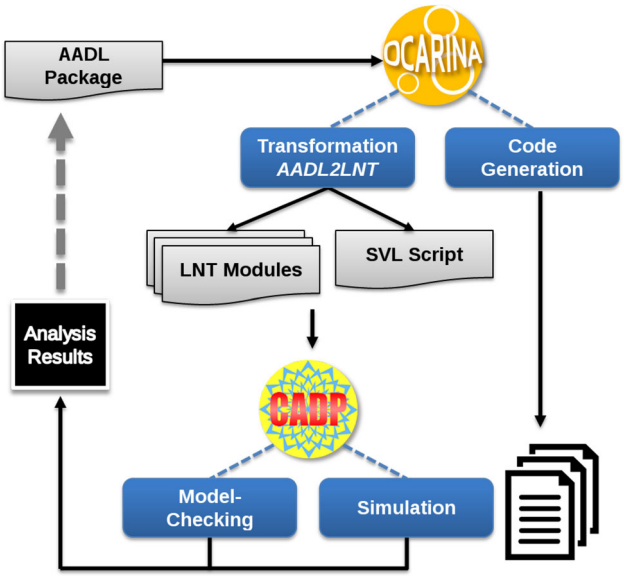

Fig. 12 Ocarina-CADP tool-chain

able with both academic and commercial licenses.[6] In addition to LNT, it supports many other input languages such as LOTOS, FSP and EXP. It also provides a scripting language SVL (Script Verification Language) [20] for the description of analysis scenarios. The toolbox offers a comprehensive set of tools for specification, interactive simulation, verification (model-checking, equivalence checking, etc.), performance evaluation, etc. To deal with complex systems, CADP provides a set of verification techniques such as the reachability analysis, on-the-fly verification and distributed verification.

4.4.1 Ocarina extension

The proposed *AADL2LNT* transformation is integrated within the Ocarina tool suite.[7] The Ocarina compiler is designed with a modular architecture distinguishing three parts: a central library (a set of routines), the frontend (for model analyses) and the backend (for model manipulations and generations). Different model manipulations are handled using the ASTs (Abstract Syntax Tree) which are the internal representation of models (AADL, annexes and other languages).

Generally described, the *AADL2LNT* model transformation is implemented in the Ocarina backend. We assume that the AADL model should be successfully analyzed on the frontend and then transformed into an AADL AST. We do not use model transformation languages for our generation. We directly apply the transformation rules on the AADL AST (without meta-model) to form a corresponding LNT AST. Then, this AST is scanned in order to produce the source code files (`*.lnt`). In addition to the LNT modules, a script file (`demo.svl`) is also generated, containing a set of operations specified in the SVL language to orchestrate the analysis phase. This file is directly generated for each AADL system (without an SVL AST).

4.4.2 CADP formal verification

The formal verification allows designers to prove that a system satisfies its requirements. The verification techniques are applied on an abstract mathematical model of the system (system state space), built according to the considered specification language semantics. In addition, the verified properties (system requirements) should be specified as graphs or temporal logic properties, using specific formalisms. Then, the verification can be performed by the analysis tools. In our work, we mainly deal with the model-checking technique, which consists in checking whether the system satisfies a given property specified with a temporal logic.

Based on the Ocarina generated outputs, a formal verification phase can be performed by the CADP toolbox using the SVL script, which guides the compilation of the LNT specification and the verification of a set of behavioral and temporal properties. The generated LNT specification is firstly compiled into an LTS (Labeled Transition System) to be explored using the CADP model checkers. To automate this operation, we include the verified properties within the SVL script. Using the SVL `property` statement, we define a set of generic properties to verify some requirements of real-time systems such as deadlock, schedulability, communication problems (e.g., data loss) and queuing problems (e.g., overflow of buffers).

We thereby drew a tool-chain providing an automatic and transparent verification of the AADL model. The transformation is carried out by the Ocarina command line, then, the generated SVL script is simply invoked to begin the verification phase with the CADP toolbox. Finally, the analysis results help designers in AADL model correction and improvement. This operation can be iteratively applied after each modification, throughout the development process, until the generation of the final application.

5 Experiments

The *AADL2LNT* transformation has been tested with various examples. In this section, we report experiments performed on two case studies: **FCS** (*Flight Control System*) and **LFR** (*Line Follower Robot*) [47], to illustrate the periodic/sporadic tasks and data/event connections model transformation and

[6] http://cadp.inria.fr/.

[7] The *AADL2LNT* transformation is integrated in the official Ocarina GitHub repository.

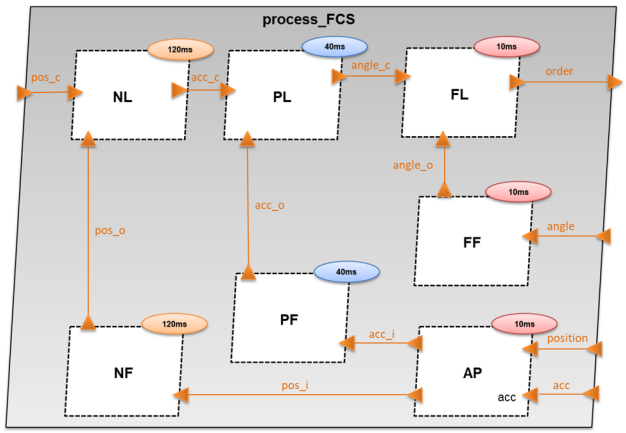

Fig. 13 **FCS** AADL model (`process` level)

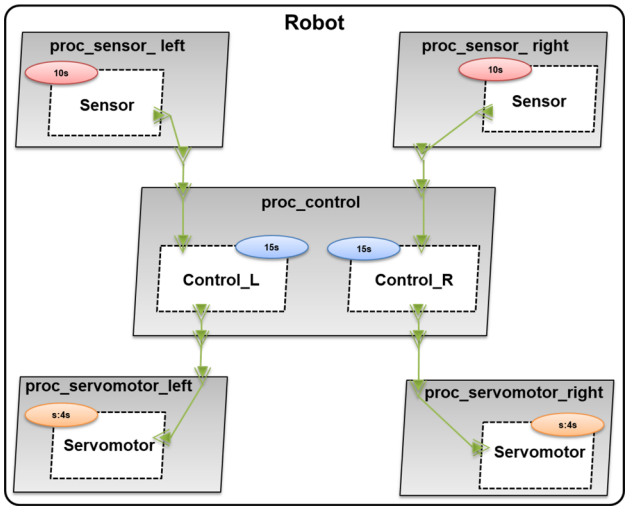

Fig. 14 **LFR** AADL model (`system` level)

formal analysis. We describe the considered AADL models; then, we expose our experimental results about the Ocarina generation and the CADP formal analysis. Finally, we discuss the usability of the analysis results and the scalability of our solution.

5.1 Modeling phase

This phase consists in defining the task models (real-time parameters, connections, etc.) and modeling the whole system with the AADL language.[8]

5.1.1 Flight control system

We consider this example to illustrate periodic tasks and data connections. **FCS** is a safety-critical avionics system for aircraft controlling. This system controls the altitude, trajectory and speed of an airplane. We consider a simplified version, composed of 7 periodic tasks ($S_{FCS} = \{\tau_{1...7}\}$) which collaborate through exchanging data, in order to send a feedback to the flight control surface.

Briefly described, the **FCS** model consists of a first subset of tasks (`FL` (Feedback Law), `FF` (Feedback Filter) and `AP` (Acceleration Position Acquisition)), which is executed at 10 ms to acquire the state of the system (angles, position, acceleration) and compute the feedback law of the system, in order to send the final order to the flight control surface. A piloting-loop subset of tasks (`PL` (Piloting Law) and `PF` (Piloting Filter)) is executed at 40 ms to determine the acceleration to apply. Finally, a navigation-loop subset of tasks (`NL` (Navigation Law) and `NF` (Navigation Filter)) is executed at 120 ms to determine the position to reach.

The corresponding **FCS** AADL model, partially depicted in Fig. 13, counts 186 lines. It consists of 7 periodic `threads` (`FL`, `PL`, `PF`, `NL`, `NF`, `AP` and `FF`) grouped in one `process` (`process_FCS`) bound to one `processor`. The model contains a set of *inter-thread* connections (declared at `process` level) and a set of *process-device* connections with 5 `devices` (declared at `system` level).

5.1.2 Line follower robot

LFR is our second case study to test sporadic tasks and event data connections. This robot is a machine that follows a black line on a white area. This system uses sensors to detect the line and control units to make movement decisions and command right and left servomotors (wheels). The robot sensors control regularly the follow-up of the black line and send information to the control units. While, the servomotors are commanded (to turn on/off) only if the robot loses the line, which consists of a non-periodic action. Thus, the servomotor behavior would be better modeled within the sporadic tasks.

The **LFR** task model ($S_{\text{LFR}} = \{\tau_{1...6}\}$) considers right/left robot sides similarly. Each side contains 3 tasks exchanging event data: a periodic task for sensing, a periodic task for controlling and a sporadic task for turning on/off the servomotor.

The corresponding **LFR** AADL model, partially depicted in Fig. 14, counts 131 lines. The model contains 6 `threads` in communication (*sensor-control* and *control-servomotor* exchanges) through event data port connections: each sensor/servomotor `thread` is contained in a `process`, while the control `threads` are grouped together in the same

[8] The **FCS** and the **LFR** AADL models are available on-line in the AADLib GitHub repository, which is a library of reusable AADLv2 models under the OpenAADL project (https://github.com/OpenAADL/AADLib).

Table 5 Case studies transformation metrics

FCS		LFR	
AADL	LNT	AADL	LNT
Source code lines		Source code lines	
188	697	131	566
Transformation		Transformation	
7 `threads`	7 `THREAD_*`s	6 `threads`	6 `THREAD_*`s
4 `devices`	4 `DEVICE_*`s	–	–
1 `process`	–	5 `processes`	–
17 data connections	12 `Data_CONNECTOR`s	10 event data connections	4 `Event_Data_CONNECTOR`s
1 `processor`	1 `SCHEDULER`	1 `processor`	1 `SCHEDULER`
1 `system`	1 `MAIN`	1 `system`	1 `MAIN`
LTS		LTS	
–	2439 states	–	673 states
	14,570 transitions		673 transitions

`process`. All the `process` components are bound to one `processor`. Other hardware components can be added to complete the model (a set of `devices` for the sensors and servomotors), yet the current software model is sufficient for our purposes.

5.2 Automatic model generation

The transformation and generation are performed by our Ocarina extension (*AADL2LNT*). For each AADL model, Ocarina generates 5 LNT modules in separate files: `*_Ports.lnt`, `*_Threads.lnt`, `*_Processor.lnt`, `*_Types.lnt` and the root module `*_Main.lnt`, with the script file `demo.svl`.

Table 5 sums up the transformation metrics of the **FCS** and **LFR** case studies. The **FCS** specification counts 697 lines and contains 25 processes. The **LFR** specification counts 566 lines and consists of 12 processes. For each case study, all the LNT processes are instantiated and synchronized at the `MAIN` level, despite the difference in the port connection types and declaration levels: in the **LFR** case study, the different event data port connections declared following the AADL component hierarchical containment, are abstracted in only 4 `Event_CONNECTOR` instances at the `MAIN` level; and in the **FCS** case study, the *process-device* connections are similarly considered as the *inter-thread* connections (12 `Data_CONNECTOR` instances).

The *AADL2LNT* transformation complexity depends mainly of the number of AADL `threads` and port connections: from models with independent `threads`, we obtain simple LNT models without the `CONNECTOR` synchronizations. While with highly connected models, the number of the required processes increases significantly.

The Ocarina automatic generation covers all the necessary steps for the transformation from AADL into LNT and eliminates its complexity, especially in the `SCHEDULER` mapping which is less generic compared with the other processes (`THREAD_*`, `DEVICE_*` and `*_CONNECTOR`) generation. For example, the **LFR** `SCHEDULER` counts 250 lines (nearly 50% of the code). Another difficulty eliminated with our *AADL2LNT* generation resides in the composition and synchronization phase: the mapping of the `MAIN` process seems tricky since we deal with a lot of process instances and gates, especially for the mapping of the port connections, in which different hierarchical declarations (at the `process` and `system` levels) should be abstracted at the `MAIN` level. For example, to map the 7 `threads` of the **FCS** case study, we should synchronize 25 process instances on 32 different gates.

5.3 Formal analysis

After the *AADL2LNT* generation, various analysis can be performed by the CADP toolbox. In our work, we use two important formal techniques: simulation (for example with the *OCIS* simulator) and model-checking. We remind that the analysis phase is described in the `demo.svl` through the following two steps, developed in the next sections.

5.3.1 State space generation

This is an imperative step to enable the verification of the LNT specifications. A translation from LNT into LOTOS is firstly applied with the *Lnt.Open*, *Lnt2LOTOS* and *Lpp* tools. Then, a generation of an LTS is performed by the *CÆSAR* compiler [24]. The LTS represents the dynamic behavior of

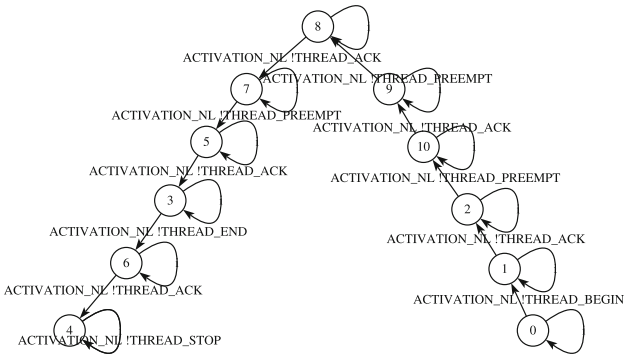

Fig. 15 The generated LTS corresponding to Listing 14

the LNT specification with a set of states and transitions (system state space). The LTSs can be explicitly manipulated with the *BCG* (Binary-Coded Graphs) tools. *BCG* is both a format for the LTS representation and a set of libraries and programs dealing with LTSs (information, display, edition, minimization, etc.).

Moreover, the LTS generation can be smartly reduced to improve the verification performance [17]. We can also personalize the LTS generation using the SVL language (hide, cut, rename labels, etc.). For example, Listing 14 and Fig. 15 represent a `divbranching` reduction [21] of the **FCS** case study, that hides all the LTS labels except the `ACTIVATION_6` gate. The resulting LTS corresponds to the activation graph of the `NL thread`, whose graph (`Main.bcg`) is drawn in Fig. 15.

Listing 14 Smart generation with SVL language

```
"Main.bcg"=  divbranching reduction of
hide all but ACTIVATION_NL in
   rename ACTIVATION_6 -> ACTIVATION_NL in
      "FCS.lnt"
   end rename
end hide;
```

The LTSs metrics of our case studies are included in Table 5. These results show the effectiveness of our contribution and the improvements achieved for our work and for the Ocarina formal verification in general:

- The **LFR** and **FCS** systems are represented with small state spaces.
- Compared to **LFR**, **FCS** has the largest state space since it counts more `threads` and connections. Note that the AADL `devices` presence only increases the transitions number without changing the activation graph.
- Regarding the state explosion problem met with our old mapping, we note a significant reduction (up to 100%) in the state space of **FCS** compared to statistics given in [43].
- Considering the scheduling level, the obtained activation graph can be compared with analysis results performed with existing schedulability analysis tools. For example, we choose the Cheddar [53] tool, since it supports AADL as input model. The generation of the **FCS** LTS can be personalized by cutting the `T_Stop` and `T_Completion` labels; thus, we obtain the activation graph counting 52 states corresponding to the same number of the context switches found by Cheddar when analyzing the **FCS** AADL model.
- Note that the Ocarina tool suite is extended in [47–49] by the generation of a Petri net model for formal analysis with the Tina tool. This work is illustrated by the same robot case study in [47]. Compared to our experimental results, we note a significant reduction in the state space metrics: about 673 states and 673 transitions for the **LFR** case study, compared to 65 527 states and 425 985 transitions for a Petri net model without a timer.

5.3.2 Verification

After the state space generation, we reach the model-checking phase. As mentioned before, we use the SVL `property` statement to specify the verified properties. This statement can be parameterized by a set of parameters, it can embed a set of verification statements such as the temporal logic verification statement that integrates a temporal logic formula. In our work, we choose the *Evaluator* model checker [40,41] and so we define each `property` formula with the temporal logic MCL (model-checking language). In addition, the *Evaluator* model checker allows the definition of macros for the temporal operators parameterized by action and/or state formulas. Thus, we use a set of predefined macros (`standard.mcl` library), such as `NEVER (R)` (there is no action sequence R) and `AFTER_1_INEVITABLE_2 (A, B)` (after an action A, the action B is inevitably reachable). In the following, we include some examples of the defined generic properties:

- `Scheduling_Test` property (Listing 15): this property indicates if a given `thread` has respected all its deadlines (absence of the `T_Error` label);
- `Is_Preempted` property (Listing 16): a `thread` may be preempted by the `SCHEDULER`, this property detects if a given `thread` has been preempted during the scheduling. The absence of all the `T_Preemption`, `T_Dispatch_Preemption` and `T_Preemption_Comple-tion` labels means that the `thread` is never preempted;
- `Connection` property (Listing 17): this property verifies if a port connection is well established, through a given port connection `AB`, after the transfer of the data

or event (a *rendezvous* on SEND_AB gate), there is at least one reading of the port content (a *rendezvous* on RECEIVE_AB gate);
- Data_Loss property (Listing 18): this property detects the loss of data through a given data port connection AB, it detects the occurrence of two successive transfers of data (*rendezvous* on SEND_AB gate), without a reading of the port content, in this case, the oldest input is overwritten by the newest one;
- Overflow_FIFO property (Listing 19): this property detects if a list (FIFO with N size) of an event/event data port is overflowed, it detects the occurrence of N+1 successive transfers of events, without any reading of the port content, in this case, the oldest input is overwritten by the N+1th one.

Listing 15 Scheduling property

```
property Scheduling_Test (ID)
   "Scheduling test of thread $ID"
is
   "Main.bcg"|= with evaluator3
   NEVER(true *. 'ACTIVATION_$ID !T_ERROR');
   expected TRUE;
end property;
```

Listing 16 Preemption property

```
property Is_Preempted (ID)
   "Preemption test of thread"
is
   "Main.bcg" |= with evaluator3
   NEVER (true *.
   'ACTIVATION_$ID !T_DISPATCH_PREEMPTION') and
   NEVER (true *.
   'ACTIVATION_$ID !T_PREEMPTION') and
   NEVER (true *.
   'ACTIVATION_$ID !T_PREEMPTION_COMPLETION');
   expected TRUE;
end property;
```

Listing 17 Port connection property

```
property Port_Connection (ID)
   "After a SNED_$ID action,
   a RECEIVE_$ID is eventually reachable"
is
   "Main.bcg" |= with evaluator3
   AFTER_1_INEVITABLE_2 ('SEND_$ID !AADLDATA',
                         'RECEIVE_$ID !AADLDATA');
   expected TRUE;
end property
```

Listing 18 Data loss property

```
property Data_Loss (ID)
 "Between two consecutive SEND_$ID actions,
  there is a RECEIVE_$ID action"
is
   "Main_$ID.bcg" =
   hide all but "SEND_$ID", "RECEIVE_$ID" in
      "Main.bcg" |= with evaluator4
   NEVER (true* . 'SEND_$ID !AADLDATA' .
          (not 'RECEIVE_$ID !AADLDATA')* .
          'SEND_$ID !AADLDATA');
   expected TRUE;
end property
```

Listing 19 FIFO property

```
property Overflow_FIFO (ID, N)
   "Between $N consecutive SEND_$ID actions,
   is a RECEIVE_$ID action"
is
   "Main_$ID.bcg" =
   hide all but "SEND_$ID", "RECEIVE_$ID" in
      "Main.bcg"
   |= with evaluator4
   NEVER (true* .
   ('SEND_$ID !AADLDATA'){$N + 1} .
   (not 'RECEIVE_$ID !AADLDATA')* .
   'SEND_$ID !AADLDATA');

   expected TRUE;
end property;
```

The defined properties allow the detection of serious problems at the model level. Since we deal with real-time systems, it is necessary to validate temporal and communication parameters such as deadlock detection, scheduling test and detection of connection failures (overflow of buffers, loss of data, broken links). We check both **LFR** and **FCS** models, and we perceive that they are well scheduled and deadlock-free. All port connections are well established. We test the **LFR** system with different FIFO sizes, and we find that the *sensor-control* queue size should be ≥ 2 to avoid the overflow problem.

5.4 Analysis results

An important issue in the AADL formal approaches concerns the usability of the obtained analysis results which are produced by the formal tools. In our work, we provide a user-friendly (simple) output form, that is easily interpreted by non-formal-expert designers.

Based on a traceable definition of the LNT specification and the SVL verified properties, we preserve information from the initial AADL model that is readily distinguished in the displayed results. The traceability is firstly assumed during the model generation: the LNT specification keeps the names of the AADL components and port connections, which are used in the naming rules of the LNT variables, gates and processes. In addition, the use of parameterized and commented properties furthers the traceability and gives understandable results. Parameters are used to represent port connections and threads by their initial AADL names. Thus, each thread or port connection can be separately verified and so failures are rapidly localized in the AADL model.

For illustration, we include an extract of the outputs of the **LFR** case study in Listing 20. The Scheduling_Test property is applied for each thread, so when the system scheduling test fails, the concerned thread (with a missed deadline) is directly localized. Similarly, the port connections are separately handled by their AADL initial names:

the Sensor_Control_R port connection corresponding to the *sensor-control* right side connection (Fig.14) is obviously identified in Listing 20 (the PASS response means that no overflow is detected in this connection while the simulation).

Listing 20 Analysis results from the **LFR** case study

```
property PROPERTY_Deadlock
PASS

property Scheduling_Test (1, TH_SERVOMOTOR_R)
 |Scheduling test of the thread TH_SERVOMOTOR_R
PASS

property Overflow_FIFO (SENSOR_CONTROL_R, 3)
 |Between 2 consecutive SEND_SENSOR_CONTROL_R
 actions, there is a RECEIVE_SENSOR_CONTROL_R one
PASS
```

5.5 Scalability discussion

The proposed tool-chain has been initially tested with a set of case studies (flight control system, pacemaker, door management system, etc.) from the AADLib library[9] to validate the correctness of the *AADL2LNT* generation. Such examples with a certain scale are easily verified in few minutes with basic machines. In addition, advanced experiments were carried out to evaluate the scalability of our solution.

In a formal context, the state space explosion is a serious issue that discourages the application of formal methods. The problem occurs when the system state space becomes too large to be verified: the number of states or transitions explodes. In our work, we deal with real-time systems based on the parallel concurrent behaviors which often lead to large state spaces. To avoid the pitfall of the state explosion problem, a set of refinements (discussed in Sect. 4.3) were applied on the LNT mapping in order to obtain small state spaces. In fact, the resulting LTS size depends on many factors related to the given AADL model at different levels (tasking model, scheduling protocol, communication, etc.). In this paper, we consider two main factors which are the number of threads and the simulation interval $H(\tau_{1...k})$.

A test suite is defined, composed of 100 AADL models, to evaluate three model families: (i) models with independent threads; (ii) models with periodic threads and data port connections; (iii) models with sporadic threads and event port connections. The test is based on a set of generalized *Producer–Consumer* system: we increase progressively the number of *Producer–Consumer* couples with different thread periods to also increase the $H(\tau_{1...k})$ value. Starting from models with only 2 threads, we reach 40 and 70 threads to be tested during thousands of units of time ($H(\tau_{1...k})$). The AADL *Producer–Consumer* models are transformed into LNT specifications using Ocarina and then compiled into LTSs using CADP.

Results Tables 6 and 7 summarize some of the obtained LTS metrics. Table 7 concerns a set of exhaustive tests of family (i): the $H(\tau_{1...k})$ value is between 100 and 30,000 units of time (row 1), and the number of threads is between 50 and 70 (column 1). Table 6 regroups results of families (ii) and (iii): the $H(\tau_{1...k})$ value is between 100 and 5000 units of time (row 1) and the number of threads is between 2 and 40 threads (column 1). For each test, we give the number of states of the generated LTS (with divbranching reduction).

In general, the LTS size grows as the number of threads and units of time ($H(\tau_{1...k})$) increases, yet each family test has some particular observations:

(i) This family includes the state spaces obtained from independent-thread AADL models. As shown in Table 7, the LTSs are quite small compared to the important number of threads. The exhaustive experiments reach 50 threads tested until the 30,000 units of time without explosion. We notice an interesting results with 50 or 60 threads, simulated during 100 units of time, which are tested in pretty small spaces (about one hundred states). Note that, beyond 70 threads, the LTSs grow exponentially and the state explosion problem is more frequent which makes models with such a scale hard to be verified.

(ii) When adding port connections, the corresponding LNT processes (CONNECTOR) are added in the obtained LNT specification which increases the size of the generated LTS. For this reason, the tests are limited to 40 threads, and beyond that, the state explosion problem is more frequent. Nevertheless, we still obtain interesting results: starting with only 29 states for 2 threads and 100 units of time, reaching 2020 states for 40 threads and 5000 units of time (Table 6).

(iii) This family includes models with sporadic threads and event or event data ports. In the *Producer–Consumer* system, the consumer thread becomes sporadic with an event port connection. Being tested in the same scope as the family (ii), the obtained LTSs of this family are much bigger and lead to some explosions (mainly with 40 threads). This is explained by the fact that additional synchronizations are required for the sporadic threads scheduling (the NOTIFICATION synchronizations between the processes SCHEDULER and CONNECTORs). Yet, we still obtain reasonable results. For example, models with 10 and 20 threads are successfully simulated with small state spaces (between 2802 and 835,786 states).

[9] https://github.com/OpenAADL/AADLib/tree/master/examples.

Table 6 State spaces results of family (ii) and (iii)

	Periodic threads with data port connections (ii)							Sporadic threads with event port connections (iii)						
	100	300	500	1000	1500	3000	5000	100	300	500	1000	1500	3000	5000
2	29	37	23	30	37	113	89	136	563	997	2052	2904	5910	9782
8	121	140	196	266	326	116	386	1590	7028	12,116	27,318	39,254	85,212	134,064
10	153	175	115	386	175	145	523	2802	9875	17,820	39,788	58,401	134,750	212,148
20	305	350	230	498	350	395	1003	7960	36,655	63,192	129,619	211,519	504,365	835,786
40	428	269	774	836	700	2290	2020	38,820	∞	∞	∞	∞	∞	∞

Table 7 State spaces results of family (i)

	100	5000	10,000	30,000
50	114	1716	2859	16,001
60	134	1804	∞	∞
70	∞	1884	2610	∞

This test suite has been carried out on a machine with high performance[10] using the 2017-J "Sophia Antipolis" CADP version. The generation of all the LNT specifications needs a few seconds, while the analysis time (LTS generation and model-checking of properties) depends evidently of the test and may require some minutes or hours. The LTS generation of family (ii) needed 28 min, which is a satisfying time for a test suite composed of 43 models counting up to 40 `threads`. From family (iii), the analysis time of a *Producer–consumer* model with 10 `threads` simulated during 1500 units of time is about few seconds (with a basic machine, it may take 4 min). While it takes about 1 h for a *Producer–consumer* model with 20 `threads` simulated during 1000 units of time from the same family (iii).

To conclude, the size of LTS depends directly on the number of `threads`, but also on other minor factors such as the scheduling mapping and the obtained activation graph: the sporadic and preemptive `threads` are more expensive (in size) than independent, periodic or non-preemptive `threads`; similarly, highly connected `threads` increase the size of the LTSs significantly. Compared to the considered subset, the proposed AADL model verification requires reasonable resources in memory and time. These experimental results are promising, showing the effectiveness of our solution to verify real-time systems with a respectable scale.

6 Related work

Since we deal with a Ravenscar compliant model, we firstly note that certain approaches have been proposed for the formal analysis of the Ravenscar systems. In general, these works aim to analyze the Ada real-time systems, such as: Kristina et al. [38] propose a formal mapping of a Ravenscar compliant runtime kernel for verification with the UPPAAL model checker; authors in [28] work on the generation of the Ada Ravenscar code from the AADL models, in which a particular data connector DBX (Deterministic Bridge Exchangers) is manually mapped in LOTOS to verify its deterministic behavior. In addition, they provide a static and dynamic semantics for the generated Ada Ravenscar in [29]; authors in [45] present a transformation of Ada Ravenscar programs using the IF timed automata. Compared to these works, we use the Ravenscar profile to apply a set of strong constraints at the model level for safety-critical real-time systems modeling. We provide an LNT mapping for a Ravenscar task model that can be completed and automated for the Ada code analysis with the CADP toolbox, which presents an important perspective. But, we currently focus on the verification of the architectural models rather than the code analysis: our Ravenscar LNT mapping is completed with architectural descriptions to transform AADL models, also it is extended to support the AADL `Behavior` annex.

The proposed *AADL2LNT* transformation is classified among AADL formal approaches. Several AADL transformations have been developed for diverse objectives. Table 8 summarizes a set of studies grouped in three families according to their target formalism: Petri nets, automata and other specification languages (e.g., process algebras). We include 18 AADL transformations with their target languages, the tools used in the transformation/formal verification and some objectives. In the following, we detail some works, aimed mainly to verify behavioral and temporal properties.

The first family concerns the Petri nets and their extensions. We mention Renault et al. work, in which an AADL subset is firstly transformed into symmetric nets in [49] and then extended into timed and colored Petri nets in [48] for the verification of temporal properties such as missed deadline or missed `thread` activation, using the Tina formal analysis tool.

The second formalism is the automata. Particularly, we note the use of the timed automata and the UPPAAL

[10] Processor Intel Xeon(R), 2.20 GHz x32, 63GB RAM, running Linux MATE 1.12.

Table 8 Summary of AADL transformation approaches

	Language	Tools		Objectives	Papers
		Transformation	Verification		
Petri nets	GSPN	ADAPT	SURF-2	Dependability analysis	[52]
	SAN	ADAPT-M	MoBIUS	Dependability analysis	[30]
	TPN	Ocarina	Tina	Temporal properties	[48,49]
Automata	Linear hybrid automata	–	TIMES	Scheduling analysis, simulation	[26]
	Event data automata	COMPASS	COMPASS	Behavioral properties, safety analysis, FDIR and performance evaluation	[10–12]
	Timed automata	–	UPPAAL	Control/data-flow reachability	[33,34]
	Timed automata	ATL	UPPAAL	Behavioral properties	[27]
	Timed automata	–	UPPAAL	AADL mode analysis	[61]
Formal languages	LUSTRE	aadl2sync	Lurette, Lesar	Simulation, behavioral properties	[32]
	BIP	OSATE	BIP framework	Behavioral properties	[15]
	TLA+	TOPCASED	model checker TLC	Temporal analysis	[50]
	Signal	OSATE	SynDEx, Polychrony	Temporal and scheduling analysis	[8,58,59]
	FIACE	TOPCASED	Tina	Simulation, behavioral properties	[7,9]
	ACSR	OSATE	VERSA	Temporal and scheduling analysis	[54,55]
	Real-Time Maude	OSATE	Maude platform	Behavioral properties	[4–6,46]
	IF	TOPCASED	–	Safety properties	[3]
	TASM	OSATE, ATL	TASM, UPPAAL	Behavioral properties	[31,57]
	Stateful Timed CSP	OSATE	PAT	Behavioral and temporal properties	[60]
	LNT (our approach)	Ocarina	CADP	Behavioral and temporal properties	[43,44]

model checker. For example, Hamdane et al. [27] describe a tool-chain from AADL into timed automata for the model-checking of behavioral properties like deadlock, liveness and reachability properties.

Bozzano et al. [10–12] propose a comprehensive platform COMPASS for the analysis of AADL models such as the requirements validation, functional verification, safety analysis, FDIR (Fault Detection, Isolation and Recovery) and performance evaluation. This approach is based on the defined SLIM language, which is an extension of the AADL language and its Error annex. The SLIM model is transformed into an EDA (event data automata) to be explored with different COMPASS tools (the NuSMV symbolic model checker, the MRMC probabilistic model checker and the RAT requirements analysis tool).

The third family transforms different AADL subsets into diverse specification languages. Our work is included among this family using the LNT language. Firstly, we note that a set of approaches addresses synchronism by using synchronous languages as target formalisms such as in [32], where authors explain how the synchronous paradigm can be used to describe asynchronous behaviors through the transformation of an AADL subset into the synchronous language Lustre. Other works deal with an AADL synchronous subset, for example, the contributions around the Polychrony framework ([8,58,59]) introduce the concept of co-modeling using an AADL subset (periodic `threads` and data port connections) for modeling and the Simulink language for the behavioral specifications. The verification is based on the transformation into the Signal synchronous language, which allows the exploration of the AADL model with the Polychrony and SynDEx tools. Yang et al. [31,57] use the same synchronous subset adding AADL modes, `Behavior` annex and offline non-preemptive scheduling policy to define a formal semantics with the TASM language. The transformation is implemented in the OSATE environment and formally verified by the Coq theorem prover, in order to verify behavioral properties (deadlock and reachability) with the TASM and UPPAAL tools. The proof is performed by equivalence checking and based on the equivalence checking of the TTS (Timed Transition System) of both the AADL and TASM models. In our proposal, we rather handle asyn-

chronous model supported by the AADL language to deal with larger AADL subsets for more realistic applications.

The transformation of AADL model into FIACRE language is addressed by Berthomieu et al. in the TOPCASED environment [7]. A second version of this work is presented in [9] dealing with an AADL synchronous subset. The verification needs a first transformation into the FIACRE language, and then the FIACRE model is compiled into an abstract timed transition system supported by the Tina tool for model-checking. This work considers the AADL model as a set of communicating threads and supports periodic/sporadic `thread` and event/data port connections, but it is restricted to the non-preemptive scheduling.

Another work [46] uses a formal real-time rewriting logic semantics, Real-Time Maude to transform an AADL subset with its `Behavior` annex to an executable semantics with the Real-Time Maude platform (an AADL simulator and LTL model checker). In addition, authors in [4–6] are motivated by the PALS pattern that reduces the design and verification of an asynchronous system with its synchronous version. They define a Synchronous AADL sub-language and provide its formal semantics in Real-Time Maude.

Chkouri et al. [15] define a translation from a significant AADL subset with its `Behavior` annex into the BIP language, and then the BIP model is transformed into a non-timed model to enable the model-checking (Aldebaran and observers tools) and the simulation with the BIP framework. This work supports periodic/sporadic `thread` and event/-data port connections, but it uses a simple scheduler without preemption.

In general, all the related works aim practically at defining a formal executable semantics for an AADL subset to allow the model-checking of behavioral and temporal properties. However, subsets, methodologies and tools are diverse. Compared to the existing approaches, we are distinguished by the following points:

- The *AADL2LNT* transformation considers both software and hardware AADL components with the consideration of a significant set of temporal and queuing standard properties: well we focus on the AADL `thread` scheduling execution and port connection mechanism with the definition of an explicit scheduler. We support the event-driven tasks, preemptive priority-based scheduling and asynchronous communications. The considered subset covers the fundamental real-time features that can be used in more realistic applications rather than synchronous and non-preemptive approaches.
- Many existing works require more than one model transformation to be connected to the analysis tools. We use LNT as target model which is a direct input language (without additional transformations) for the CADP toolbox. This gateway allows the exploitation of the CADP toolbox that offers a variety of formal methods (model and equivalence checking, simulation, etc.) and provides mature solutions for the state space explosion problem (smart state space reductions, on-the-fly verification, etc.).
- For the soundness of the AADL transformation, [9,57] propose a semantics preservation proof based on the formalization of an TTS semantics for a restricted AADL subset. Then, an equivalence relation is checked with the corresponding TTS of the target language using the Coq theorem prover. However, the AADL-TTS semantics definition is supposed to be correct without preservation proving. This semantics gap concerns all the AADL formal transformations. Due to the informal semantics of the AADL language, we cannot directly prove the equivalence between the AADL semantics and the formal semantics of the target formalism. In our work, we propose an LNT mapping for a task model compliant with the Ravenscar profile that can be reused on other architectural transformations. We use a standard task model as pivot representation: the AADL semantics is abstracted as a task model which reduces semantic ambiguity in the transformation.
- The *AADL2LNT* transformation is the result of several adjustments (at the AADL subset) and refinements (in the LNT mapping) to obtain formal specifications exploitable with the verification tools. Based on the encapsulation of the temporal calculations within the `SCHEDULER`, the synchronizations between the processes of the LNT specification are restricted to a set of enumerated labels (activation orders, data labels, etc.), which allows to reduce significantly the generated LTSs at the analysis phase. The performed scalability study proves the efficiency and the applicability of our approach in the verification of real-time systems with a respectable scale.
- A majority of works uses the OSATE (eclipse plug-in) to implement the AADL transformation. In our work, we opt to Ocarina which is 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 for scheduling analysis and Bound-T for WCET analysis. 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.
- The Ocarina-CADP tool-chain is totally automated and transparent for the transformation and verification. The verification phase is ensured by the SVL script allowing the exploration of the system state space and the model-checking of a set of generic temporal and behavioral properties.

7 Conclusions and future work

In this paper, we reported our experience in the context of real-time systems. We mainly presented a formal mapping for real-time task model using the LNT language. This proposal has been applied and tested in an MDE approach based on the AADL modeling language. We presented an automatic transformation from AADL models into LNT specifications, implemented in the Ocarina tool suite. This generation allows the formal verification of the AADL model with the CADP toolbox. Our work has been illustrated with various real-time systems. We discussed analyses of two examples *Flight control system* and *Line follower robot* and we tested the scalability of our contribution using a family of generalized *Producer–Consumer* systems.

Throughout this paper, we detailed and justified the use of LNT to specify real-time features. We provided a formal executable semantics considering mainly the scheduling execution and communication which are indispensable for a useful analysis of real-time systems. Our proposition brings significant results face to formal verification challenges (analysis time and state space explosion) which is encouraging for more advanced mapping and analysis.

At this level, a fundamental step is achieved. As future work, the scheduling mapping can be extended to support multi-core scheduling. Another direction concerns the networking aspect and the consideration of the AADL `bus` and `virtual bus` components.

Acknowledgements We would like to thank the CADP team (Hubert Garavel, Frédéric Lang and Wendelin Serwe) for their help in using the LNT language and CADP toolbox.

References

1. AS5506A: SAE Architecture Analysis and Design Language (AADL) Version 2.0 (2009)
2. AS5506/2: SAE Architecture Analysis and Design Language (AADL) Annex Volume 2 (2011)
3. Abdoul, T., Champeau, J., Dhaussy, P., Pillain, P.Y., Roger, J.: AADL execution semantics transformation for formal verification. In: 13th IEEE International Conference on Engineering of Complex Computer Systems, pp. 263–268. IEEE (2008)
4. Bae, K., Ölveczky, P.C., Al-Nayeem, A., Meseguer, J.: Synchronous AADL and its formal analysis in Real-Time Maude. In: Qin, S., Qiu, Z. (eds.) Formal Methods and Software Engineering, pp. 651–667. Springer, Berlin (2011)
5. Bae, K., Ölveczky, P.C., Meseguer, J.: Definition, semantics, and analysis of multirate synchronous AADL. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014: Formal Methods, pp. 94–109. Springer, Cham (2014)
6. Bae, K., Ölveczky, P.C., Meseguer, J., Al-Nayeem, A.: The SynchAADL2Maude tool. In: de Lara, J., Zisman, A. (eds.) Fundamental Approaches to Software Engineering, pp. 59–62. Springer, Berlin (2012)
7. Berthomieu, B., Bodeveix, J.P., Chaudet, C., Dal Zilio, S., Filali, M., Vernadat, F.: Formal verification of AADL specifications in the TOPCASED environment. In: Kordon, F., Kermarrec, Y. (eds.) Reliable Software Technologies—Ada-Europe 2009, pp. 207–221. Springer, Berlin (2009)
8. Besnard, L., Bouakaz, A., Gautier, T., Le Guernic, P., Ma, Y., Talpin, J.P., Yu, H.: Timed behavioural modelling and affine scheduling of embedded software architectures in the AADL using Polychrony. Sci. Comput. Program. **106**, 54–77 (2015)
9. Bodeveix, J.P., Filali, M., Garnacho, M., Spadotti, R., Yang, Z.: Towards a verified transformation from AADL to the formal component-based language FIACRE. Sci. Comput. Program. **106**, 30–53 (2015). **(Special Issue: Architecture-Driven Semantic Analysis of Embedded Systems)**
10. Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M.: The COMPASS approach: correctness, modelling and performability of aerospace systems. In: Buth, B., Rabe, G., Seyfarth, T. (eds.) Computer Safety, Reliability, and Security, pp. 173–186. Springer, Berlin (2009)
11. Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M.: Safety, dependability and performance analysis of extended AADL models. Comput. J. **54**(5), 754–775 (2010)
12. Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M., Wimmer, R.: A model checker for AADL. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification, pp. 562–565. Springer, Berlin (2010)
13. Burns, A.: The Ravenscar profile. ACM SIGAda Ada Lett. **19**(4), 49–52 (1999)
14. Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., Lang, F., McKinty, C., Powazny, V., Serwe, W., Smeding, G.: Reference Manual of the LNT to LOTOS Translator. Technical report (2018)
15. Chkouri, M.Y., Robert, A., Bozga, M., Sifakis, J.: Translating AADL into BIP-application to the verification of real-time systems. In: Chaudron, M.R.V. (ed.) Models in Software Engineering, pp. 5–19. Springer, Berlin (2009)
16. Courtiat, J.P., Santos, C.A., Lohr, C., Outtaj, B.: Experience with RT-LOTOS, a temporal extension of the LOTOS formal description technique. Comput. Commun. **23**(12), 1104–1123 (2000)
17. Crouzen, P., Lang, F.: Smart reduction. In: Giannakopoulou, D., Orejas, F. (eds.) Fundamental Approaches to Software Engineering, pp. 111–126. Springer, Berlin (2011)
18. Feiler, P., Gluch, D.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language. SEI Series in Software Engineering. Pearson Education, London (2012)
19. Garavel, H., Hautbois, R.P.: An experiment with the LOTOS formal description technique on the flight warning computer of airbus 330/340 aircrafts. In: Proceedings of the first AMAST International Workshop on Real-Time Systems. Citeseer (1993)
20. Garavel, H., Lang, F.: SVL: a scripting language for compositional verification. In: Kim, M., Chin, B., Kang, S., Lee, D. (eds.) Formal Techniques for Networked and Distributed Systems, pp. 377–392. Springer, Boston (2001)
21. Garavel, H., Lang, F., Mateescu, R.: Compositional verification of asynchronous concurrent systems using CADP. Acta Inform. **52**(4), 337–392 (2015)
22. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. Int. J. Softw. Tools Technol. Transf. **15**(2), 89–107 (2013)
23. Garavel, H., Lang, F., Serwe, W.: From LOTOS to LNT. In: Katoen, J.P., Langerak, R., Rensink, A. (eds.) ModelEd, TestEd, TrustEd: Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday, pp. 3–26. Springer, Cham (2017)
24. Garavel, H., Serwe, W.: State space reduction for process algebra specifications. Theor. Comput. Sci. **351**, 131–145 (2006)
25. Geniet, D., Dubernard, J.P.: Scheduling hard sporadic tasks with regular languages and generating functions. Theor. Comput. Sci. **313**(1), 119–132 (2004)

26. Gui, S., Luo, L., Li, Y., Wang, L.: Formal schedulability analysis and simulation for AADL. In: 2008 International Conference on Embedded Software and Systems, pp. 429–435. IEEE (2008)
27. Hamdane, M.E., Chaoui, A., Strecker, M.: From AADL to timed automaton-a verification approach. Int. J. Softw. Eng. Appl. **7**(4), 115–126 (2013)
28. Hamid, I., Najm, E.: Real-time connectors for deterministic data-flow. In: 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007), pp. 173–182. IEEE (2007)
29. Hamid, I., Najm, E.: Operational semantics of Ada Ravenscar. In: Kordon, F., Vardanega, T. (eds.) Reliable Software Technologies—Ada-Europe 2008, pp. 44–58. Springer, Berlin (2008)
30. Hecht, M., Lam, A., Vogl, C.: A tool set for integrated software and hardware dependability analysis using the architecture analysis and design language (AADL) and error-model annex. In: 2011 16th IEEE International Conference on Engineering of Complex Computer Systems, pp. 361–366 (2011)
31. Hu, K., Zhang, T., Yang, Z., Tsai, W.T.: Exploring AADL verification tool through model transformation. J. Syst. Archit. **61**(3), 141–156 (2015)
32. Jahier, E., Halbwachs, N., Raymond, P., Nicollin, X., Lesens, D.: Virtual execution of AADL models via a translation into synchronous programs. In: Proceedings of the 7th ACM and IEEE International Conference on Embedded Software, pp. 134–143. ACM, New York, NY, USA (2007)
33. Johnsen, A., Lundqvist, K., Hanninen, K., Pettersson, P., Torelm, M.: AQAF: an architecture quality assurance framework for systems modeled in AADL. In: 2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA), pp. 31–40. IEEE (2016)
34. Johnsen, A., Lundqvist, K., Pettersson, P., Jaradat, O.: Automated verification of AADL-specifications using UPPAAL. In: 2012 IEEE 14th International Symposium on High-Assurance Systems Engineering, pp. 130–138. IEEE (2012)
35. Lasnier, G., Zalila, B., Pautet, L., Hugues, J.: Ocarina: an environment for AADL models analysis and automatic code generation for high integrity applications. In: Kordon, F., Kermarrec, Y. (eds.) Reliable Software Technologies—Ada-Europe 2009, pp. 237–250. Springer, Berlin (2009)
36. Léonard, L., Leduc, G.: A formal definition of time in LOTOS. Form. Asp. Comput. **10**(3), 248–266 (1998)
37. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM **20**(1), 46–61 (1973)
38. Lundqvist, K., Asplund, L.: A Ravenscar-compliant run-time kernel for safety-critical systems. Real Time Syst. **24**(1), 29–54 (2003)
39. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What industry needs from architectural languages: a survey. IEEE Trans. Softw. Eng. **39**(6), 869–891 (2013)
40. Mateescu, R., Sighireanu, M.: Efficient on-the-fly model-checking for regular alternation-free mu-calculus. Sci. Comput. Program. **46**(3), 255–281 (2003). **(Special issue on Formal Methods for Industrial Critical Systems)**
41. Mateescu, R., Thivolle, D.: A model checking language for concurrent value-passing systems. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008: Formal Methods, pp. 148–164. Springer, Berlin (2008)
42. Mkaouar, H.: A formal approach for real-time systems engineering. Ph.D. thesis, University of Sfax, Tunisia (2019)
43. Mkaouar, H., Zalila, B., Hugues, J., Jmaiel, M.: From AADL model to LNT specification. In: de la Puente, J.A., Vardanega, T. (eds.) Reliable Software Technologies—Ada-Europe 2015, pp. 146–161. Springer, Cham (2015)
44. Mkaouar, H., Zalila, B., Hugues, J., Jmaiel, M.: An Ocarina extension for AADL formal semantics generation. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, pp. 1402–1409. ACM, New York, NY, USA (2018)
45. Ober, I., Halbwachs, N.: On the timed automata-based verification of Ravenscar systems. In: Kordon, F., Vardanega, T. (eds.) Reliable Software Technologies—Ada-Europe 2008, pp. 30–43. Springer, Berlin (2008)
46. Ölveczky, P.C., Boronat, A., Meseguer, J.: Formal semantics and analysis of behavioral AADL models in Real-Time Maude. In: Hatcliff, J., Zucca, E. (eds.) Formal Techniques for Distributed Systems, pp. 47–62. Springer, Berlin (2010)
47. Renault, X.: Mise en œuvre de notations standardisées, formelles et semi-formelles dans un processus de développement de systemes embarqués temps-réel répartis. Ph.D. thesis, Université Pierre et Marie Curie-Paris VI (2009)
48. Renault, X., Kordon, F., Hugues, J.: Adapting models to model checkers, a case study: analysing AADL using Time or Colored Petri Nets. In: 2009 IEEE/IFIP International Symposium on Rapid System Prototyping, pp. 26–33. IEEE (2009)
49. Renault, X., Kordon, F., Hugues, J.: From AADL architectural models to Petri Nets: Checking model viability. In: 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), pp. 313–320. IEEE (2009)
50. Rolland, J.F., Bodeveix, J.P., Filali, M., Chemouil, D., Thomas, D.: Modes in asynchronous systems. In: 13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008), pp. 282–287. IEEE (2008)
51. RTCA/DO-333: Formal Methods Supplement to DO-178C and DO-278A (2011)
52. Rugina, A., Kanoun, K., KaÃćniche, M.: The ADAPT tool: From AADL architectural models to stochastic Petri nets through model transformation, pp. 85–90 (2008)
53. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Cheddar: a flexible real time scheduling framework. In: ACM SIGAda Ada Letters, vol. 24, pp. 1–8. ACM (2004)
54. Sokolsky, O., Lee, I., Clarke, D.: Schedulability analysis of AADL models. In: Proceedings of the 20th International Conference on Parallel and Distributed Processing, pp. 179–179. IEEE Computer Society, Washington, DC, USA (2006)
55. Sokolsky, O., Lee, I., Clarke, D.: Process-algebraic interpretation of AADL models. In: Kordon, F., Kermarrec, Y. (eds.) Reliable Software Technologies—Ada-Europe 2009, pp. 222–236. Springer, Berlin (2009)
56. Vyatkin, V.: Software engineering in industrial automation: state-of-the-art review. IEEE Trans. Ind. Inform. **9**(3), 1234–1249 (2013)
57. Yang, Z., Hu, K., Ma, D., Bodeveix, J.P., Pi, L., Talpin, J.P.: From AADL to Timed Abstract State Machines: A Verified Model Transformation, pp. 42–68. Elsevier, Amsterdam (2014)
58. Yu, H., Ma, Y., Gautier, T., Besnard, L., Le Guernic, P., Talpin, J.P.: Polychronous modeling, analysis, verification and simulation for timed software architectures. J. Syst. Archit. **59**(10), 1157–1170 (2013)
59. Yu, H., Ma, Y., Gautier, T., Besnard, L., Talpin, J.P., Le Guernic, P., Sorel, Y.: Exploring system architectures in AADL via Polychrony and SynDEx. Front. Comput. Sci. **7**(5), 627–649 (2013)
60. Zhang, F., Zhao, Y., Ma, D., Niu, W.: Formal verification of behavioral AADL models by stateful timed CSP. IEEE Access **5**, 27421–27438 (2017)
61. Zhang, Y., Dong, Y., Zhang, Y., Zhou, W.: A study of the AADL mode based on timed automata. In: 2011 IEEE 2nd International Conference on Software Engineering and Service Science, pp. 224–227. IEEE (2011)