Towards a verified transformation from AADL to the formal component-based language FIACRE

Jean-Paul Bodeveix, M Filali, Manuel Garnacho, Régis Spadotti, Zhibin Yang

Towards a verified transformation from AADL to the formal component-based language FIACRE

Jean-Paul Bodeveix [a], Mamoun Filali [a], Manuel Garnacho [a], Régis Spadotti [a], Zhibin Yang [a,b]

[a] *IRIT-CNRS, Université de Toulouse, Toulouse, France*
[b] *College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China*

A B S T R A C T

During the last decade, AADL is an emerging architecture description languages addressing the modeling of embedded systems. Several research projects have shown that AADL concepts are well suited to the design of embedded systems. Moreover, AADL has a precise execution model which has proved to be one key feature for effective early analysis.
In this paper, we are concerned with the foundational aspects of the verification support for AADL. More precisely, we propose a verification toolchain for AADL models through its transformation to the FIACRE language which is the pivot verification language of the TOPCASED project: high level models can be transformed to FIACRE models and then model-checked. Then, we investigate how to prove the correctness of the transformation from AADL into FIACRE and present related elementary ingredients: the semantics of AADL and FIACRE subsets expressed in a common framework, namely timed transition systems. We also briefly discuss experimental validation of the work.

0. Introduction

Today, it is acknowledged that the design phase of critical systems is an important challenge. In this context, the Aerospace Valley World Competitiveness Cluster AESE [3] (Aéronautique Espace et Systèmes Embarqués) together with several aerospace and spatial actors from industry and academy have joined their efforts to develop methods and tools for the new generation of safe software. The TOPCASED [43] project has been one of the main projects within this initiative.

In order to address architectural aspects of the design phase, the TOPCASED project has adopted the AADL (Architecture and Analysis Design Language) language. AADL is an architecture description language which addresses both software and hardware aspects of the system. It has been used especially in the avionics domain and is now a standard of the SAE [38].

A safety-critical system is often required to pass stringent qualification and certification processes before its deployment and provide sufficiently strong evidence of its correctness. Moreover, our aim is also to go beyond usual schedulability analysis (Rate Monotonic Analysis, RMA for short) and consider timed-behavior analysis. For this purpose, an AADL model is often transformed to another formal model for verification and analysis. Examples of such transformations are numerous: translations to Behavior Interaction Priority (BIP) [11], to TLA+ [35], to real-time process algebra ACSR [41], to IF [1], to Real-Time Maude [33], to Lustre [28], to Polychrony [32], etc. The goal of such a translation is to reuse existing verification and analysis tools and their formal model of computation and communication for the purpose of validating the AADL models.

E-mail address: filali@irit.fr (M. Filali).

http://dx.doi.org/10.1016/j.scico.2015.03.003

Within the TOPCASED project, the FIACRE language has been introduced as an intermediate formalism between high-level modeling languages (AADL, SDL, UML, etc.) and verification engines (CADP, TINA). We verify functional and real-time properties such as schedulability and timeliness properties of AADL models through their transformation to FIACRE. Thus, we present here the transformation from AADL to FIACRE.

In this paper, we are concerned with the foundational aspects of the verification support for AADL. More precisely, we propose a verification toolchain for AADL models through its transformation to the FIACRE language [16] which is the pivot verification language of the TOPCASED project: high level models can be transformed to FIACRE models and then model-checked. Then, we investigate how to prove the correctness of the toolchain and present related elementary ingredients: the semantics of AADL and FIACRE subsets expressed in a common framework, namely timed transition systems (TTS). Although all the material presented in this paper has been experimentally applied and validated, we remark that a full AADL verification environment is a long-term goal.

Our work should be considered as a contribution to the study of the following points:

- The choice of concepts, techniques and tools for expressing transformations of real-time models and their validation.
- The architecture of an AADL verification toolchain, based on the introduction of the extension FIACRE* of FIACRE and of a real-time library used to encode AADL execution model.
- The semantics of a subset of the AADL execution model and of the target language FIACRE.

The rest of this paper is organized in two parts: Part 1 is dedicated to the verification of AADL models by transformation to FIACRE. Section 1 introduces the main languages we are interested in: AADL and FIACRE. Section 2 gives an overview of the translation process from AADL to FIACRE. Part 2 is dedicated to semantics aspects related to the transformation of Part 1. Section 3 is an overview of the semantics domains that will be used. Section 4 gives the FIACRE kernel mechanization. Section 5 details the mechanization of the considered AADL subset. Section 6 discusses how to verify the correctness of a transformation by a bisimulation. After discussing some related works, we draw a conclusion.

Part 1. AADL to FIACRE transformation

This part describes some elements of the tool chain aiming at verifying AADL models. It is based on an existing real-time model checking toolbox taking as input FIACRE models. For this purpose, AADL models are transformed to FIACRE models, thus defining a formal semantics for AADL. In the following, we first present the subset of AADL accepted by the transformation. Then, we present the FIACRE language, which is the entry point for the model checking toolbox, and its extension FIACRE* which provides genericity and iterated constructs used to define a real-time library of AADL protocols introduced by the AADL execution model. We conclude this part with a presentation of a translation from AADL to FIACRE.

1. AADL and FIACRE

This section presents an overview of the AADL language and of the subset we consider for model-checking purpose. Then, the target language FIACRE is introduced together with its extension FIACRE*.

1.1. AADL

In this section, we present the AADL hardware and software component categories. Then, we elaborate the language subset that we will consider for model checking.

1.1.1. Overview of the AADL *language*

AADL describes a system as a hierarchy of software and hardware components. It offers a set of predefined component categories as follows:

- Software components: thread, subprogram, data and process.
- Hardware components: processor, memory, bus, device, virtual processor and virtual bus.
- System components which represent composite sets of software and hardware components.

A component is described by its type and its implementation. The type specifies the component's external interface in terms of features. Features can be ports, server subprograms or data accesses depending on the chosen communication paradigm. Implementations specify the internal structure of the components in terms of a set of subcomponents, their connections, modes that represent operational states of components, and properties that support specialized architecture analysis.

However, system behaviors do not only rely on the structure defined by the components described above but also on the runtime environment (like operating system or virtual machine) [15]. AADL offers an execution model that covers most of the runtime needs of real-time systems through (1) a set of execution model attributes that can be attached to each AADL declaration, such as thread dispatch protocols, communication protocols, scheduling policies, mode change protocols,

```
thread implementation t_sender.i
  subcomponents
    v: data V.i;
  annex behavior_specification {**
    states
      st : initial complete state;
      sf : complete state;
    transitions
      st -[ on a'count = 0 ]-> st {
        v.b := true;
        d!(v);
        computation(1ms,2ms);
      };
      st -[ a?(true) ]-> sf;
      st -[ a?(false) ]-> st { v.b := true; d!(v); };
      sf -[ on a'count = 0 ]-> sf {
        v.b := false;
        d!(v);
        computation(1ms,2ms);
      };
      sf -[ a?(false) ]-> st;
      sf -[ a?(true) ]-> sf { v.b := false; d!(v); };
**};
end t_sender.i;
```

Fig. 1. Behavior annex of the sender in the alternating bit protocol.

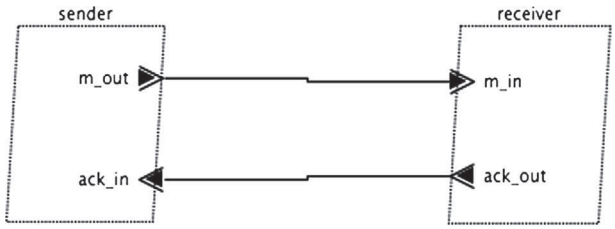

Fig. 2. The alternating bit protocol in AADL.

partition mechanisms, etc.; and (2) an execution model that uses these attributes to describe the runtime behavior of AADL models.

Moreover, the behavior annex [39] describes more precisely the behaviors of threads and subprograms. The behavior annex has an independent syntax and semantics. As an example, Fig. 1 illustrates the use of the behavior annex to describe the sender thread activity of the alternating bit protocol [30]. The protocol sends a tagged boolean message on channel d and waits for an acknowledgment on channel a. It transits from the states st to sf and conversely if the awaited acknowledged tag is received. If no acknowledge has been received the message is sent again. This operation is supposed to take some duration between 1 and 2 milliseconds. This behavior is repeated at each period of the thread.

1.1.2. The considered subset of AADL

AADL execution model has synchronous and asynchronous aspects [38,13,12]. A synchronous execution model is obtained by considering logically synchronized periodic threads communicating through registers (or shared variables) at fixed instants. In the asynchronous one, it is also possible to raise events, to specify sporadic and aperiodic threads, communication through shared variables, and remote procedure calls, etc. In this paper, for model checking purposes, we consider a subset of the software part of AADL, illustrated by Fig. 2.

Periodic & sporadic threads AADL supports the classical thread dispatch protocols: *Periodic*, *Sporadic*, *Aperiodic*, *Timed*, *Hybrid* and *Background*. *Periodic* and *Sporadic* dispatch protocols are considered in this paper. Several properties can be assigned to these threads, such as: a period given by the *Period* property in the form of Period $\Rightarrow$ 100 ms, execution time through the *Compute_Execution_Time* property or the `computation(BCET, WCET)` statement of the behavioral annex, and *Deadline*. By default, when the deadline is not specified it equals the period. A thread may also be assigned a *Priority* that is used by the scheduling protocol, supposed to be fixed priority.

Ports A thread has input ports and output ports for receiving and sending messages. AADL defines three types of ports: data, event and event data ports. Event and event data ports support queueing buffers, urgency can be associated to such ports, but data ports only keep the latest data.

Port-based communications Port connections link ports, such as data, event and event data ports, to enable the exchange of messages among components. In order to ensure deterministic data communication between the data ports of periodic threads, AADL offers two communication protocols: *Immediate* and *Delayed*. For an immediate connection, the execution of the recipient thread is suspended until the sending thread completes its execution when the dispatches of sender and receiver threads are simultaneous. For a delayed connection the output of the sending thread is not transferred until the sending thread deadline, typically the end of the period. Note that they have not necessarily the same period, which allows over-sampling and under-sampling. A potentially nondeterministic model can use shared data or event-based communications where messages may be sent at any time during the execution of a thread. Such messages may be buffered until the *Queue_Size* limit associated to in event (or event data) ports. Overflows can be managed according to the *Overflow_Handling_Protocol* by either deleting the newest or the oldest message, or by signaling an error.

Shared data AADL components can communicate through shared data. Such data is declared in data subcomponents and linked to threads via data ports. The access to shared data can be controlled by the *Concurrency Control Protocol* property so that exclusive access can be granted to some threads. However, this declaration is not supported by our translation. The most pessimistic behavior is considered where interleaving between threads of identical priority may occur at any time.

Behavior annex AADL does not support the expression of the precise behavior of threads, which is supposed to be defined using the implementation language (C, Ada) and is thus hard to analyze. This behavior is abstracted by its WCET in order to perform timing analysis for example. In order to allow more precise data dependent analyses, the behavior annex has been proposed. This annex describes through an extended transition system basic computation and port accesses, thus describing an abstract view of the real behavior of a subprogram or a thread. The behavior annex can thus be nondeterministic and can contain WCET information related to fully abstracted computations.

1.2. FIACRE

FIACRE is a formal specification language designed to represent both behavioral and real-time aspects of concurrent systems. It has been designed to support efficient model checking of linear or branching time properties through its translation to Time Petri Nets. FIACRE defines two basic concepts:

- Processes that describe the behavior of sequential components. A process is defined by a set of control states (or locations), each associated with a piece of program built from deterministic constructs available in classical programming languages or nondeterministic constructs for specification purposes.
- Components that describe the composition of processes. A component is defined as a parallel composition of components and/or processes communicating through synchronous communication channels [24] (component ports) and through shared variables. The notion of component also allows to restrict the access mode and visibility of shared variables and ports, to associate timing constraints (time intervals) with communications, and to define priority between communication events.

A more complete presentation of FIACRE can be found in the companion paper [44].

1.3. FIACRE*

FIACRE* is a syntactic extension of FIACRE designed to help writing generic libraries of protocols as for example communication and scheduling protocols used by the AADL runtime. It helps also to ease a one to one translation of high level languages to FIACRE. These extensions cover the following points:

- Genericity: processes and components can be parameterized by types, constants or type constructors.
- Arrays of ports: a process or component can be parameterized by an indexed family of ports thus allowing to connect it through one-to-one connectors with an unknown number of processes.
- Indexed selection and indexed parallel operator: these n-ary FIACRE operators are extended to support an undefined number of statements (resp. sub-components).
- Universal and existential quantification: this is the indexed counterpart of the binary conjunction and disjunction operators.
- Multiplexed ports: this extension allows to pass as effective parameter a set of ports when a unique port is required. Synchronization will be performed with one of these ports.

Note that, indexed constructs are close to the one introduced in CSP [24].

2. From AADL to FIACRE

In this section, we present an *overview* of a *Model to Text* translation of a subset of AADL to FIACRE*. More specifically, this translation takes as input an instance (model) of the AADL metamodel and outputs a FIACRE* source file. First, we describe

Table 1
Overview of structural component mapping between AADL and FIACRE*.

AADL	FIACRE*
system	component
process	∅
thread	process
event data port	port + variable
data port	variable
data	record

Table 2
Overview of type mapping between AADL and FIACRE*.

AADL	FIACRE*
behavior:integer	int
behavior::bool	bool
Data	record

the structural and behavioral aspects of the translation. Then we present a library of FIACRE* components to model the AADL runtime. Finally, we conclude with a brief presentation of a prototype implementation for this translation.

2.1. Overview of the translation

AADL systems are composed of a variety of different components nested within one another. The FIACRE* language also offers components, though more primitive than their AADL counterparts. In the considered AADL subset, a typical system is represented as a composition of processes and threads. Communication is achieved through ports. Shared data access between components is also supported. Table 1 summaries the structural mapping between AADL components and FIACRE* components. Note that the translation does not preserve the hierarchy of the component composition. For instance, the notion of AADL process is absent in the FIACRE* translation. Consequently, the resulting FIACRE* system can be seen as a flattened AADL system where only end points are preserved.

In the remaining of this section, we describe succinctly the translation for the main AADL components of the considered subset, namely Data, Thread and System.

Data components are translated to a record type in FIACRE*. Each subcomponent in a Data component is mapped to a field in the resulting record and the translation continues recursively on each Data subcomponent. Mapping of types between both languages is summarized in Table 2.

A Thread is translated to a FIACRE* process. Event ports, shared and local variables declared in the Thread are somewhat preserved during the translation. In Section 2.2.1 we give a more detailed description of this translation.

Finally, a System instance is translated to a single Component (see Section 2.2.3) in FIACRE* mainly composed of:

- port declarations with timing constraints
- variable declarations
- parallel composition of processes

2.2. The translation

Now that we know more about the mapping of components from AADL to FIACRE*, we shall detail how the inner parts of AADL components are translated. Note that both structural and behavioral aspects of AADL components are taken into account. In the following, we only consider Thread and System components because every component in-between is not preserved by the translation process.

2.2.1. Threads

As mentioned before, an AADL Thread is translated to a FIACRE* process. However, few details have been given regarding the correspondence between a Thread interface and implementation with the resulting process. First, we present the translation of the interface of a Thread (mainly composed of port declarations and dispatch protocol) then we give an overview of the translation of its implementation, namely the Behavior Annex.

A process header is composed of port declarations followed with argument declarations. Arguments may be passed by value or by reference, thus enabling shared variables. Note that different translation rules apply when the dispatch protocol of a thread is either periodic or sporadic. Fig. 3 describes the port mappings for a thread in each case. In the following, we shall mainly comment on Fig. 3 and emphasize the differences between both dispatch protocols.

On the one hand, consider the first mapping for the periodic thread. Each outgoing event (data) port is mapped to both a port and a shared variable. The port is used to perform synchronization whereas the shared variable contains the data to

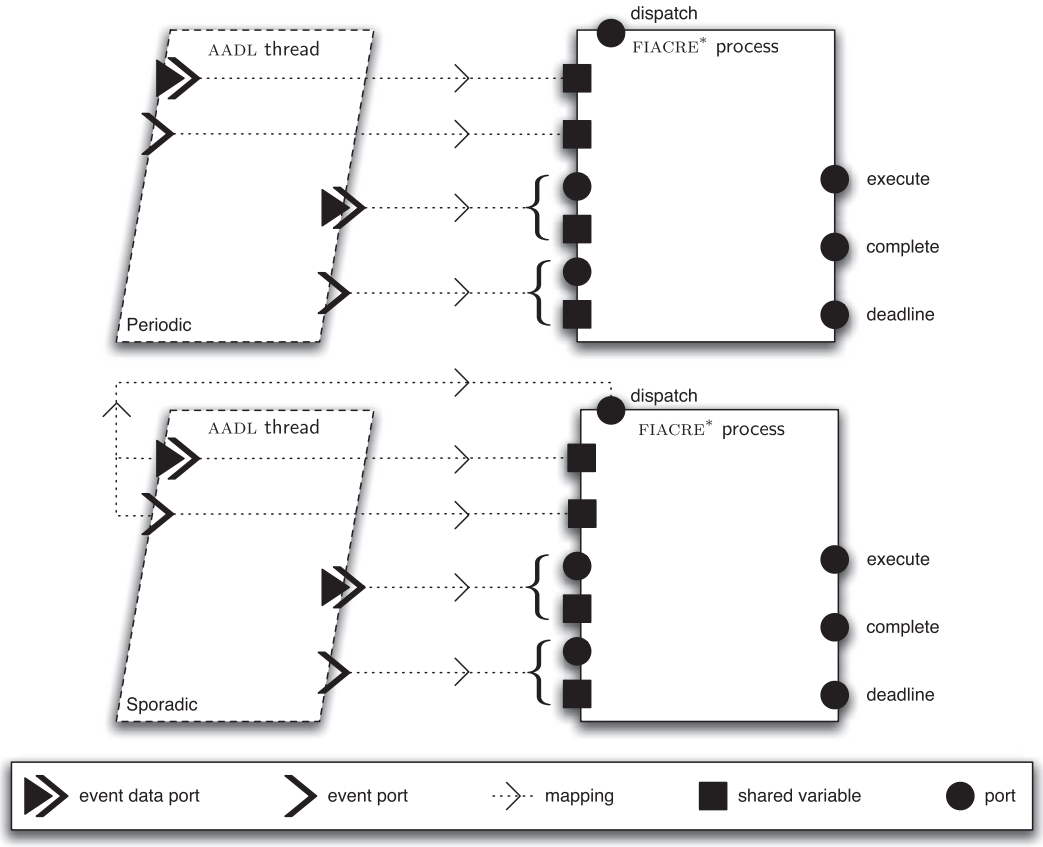

Fig. 3. Ports mapping for *periodic* and *sporadic* threads.

be transmitted. However, incoming event (data) ports are only represented with shared variables. This is justified by the fact that incoming data is only available at dispatch time (following AADL semantics), thus there is no need to synchronize port communication within the process itself. Note that the synchronization shall take place at the global level via *port controllers*. In addition, the process interface is extended with four additional ports to conform with the AADL thread execution model where:

- d: *dispatch* event port
- e: *execute* event port
- c: *complete* event port
- dl: *deadline* event port

On the other hand, the translation procedure is slightly different when dealing with *sporadic* Thread since a dispatch event may be triggered by a communication on any incoming event (data) port. As a result, the dispatch port d carries the identity of the incoming event (data) port. This is modeled in FIACRE* by giving a type τ to the port d where τ is defined as the sum of all incoming event (data) ports. For instance, given incoming event and event data ports e_1, e_2, ed_2, ed_3 the type τ is defined as union type $e_1 \mid e_2 \mid ed_2 \mid ed_3$ end.

Finally, each data port (incoming or outgoing), independently of the dispatch protocol, is mapped to a shared variable. This case is omitted in Fig. 3.

2.2.2. Behavior annex

The Behavior Annex (BA) allows to attach a behavioral specification to AADL components. This specification is mainly expressed with a state/transition automaton decorated with actions and communications (see Fig. 1). In the remaining of this section, we present an overview of the translation of this automaton.

Local variables In addition to data subcomponents, the BA offers the possibility to declare local variables. Thus, each data subcomponent and each local variable in the BA is mapped to a local process variable (*var* declaration in FIACRE*). The type of each variable is given by Table 2.

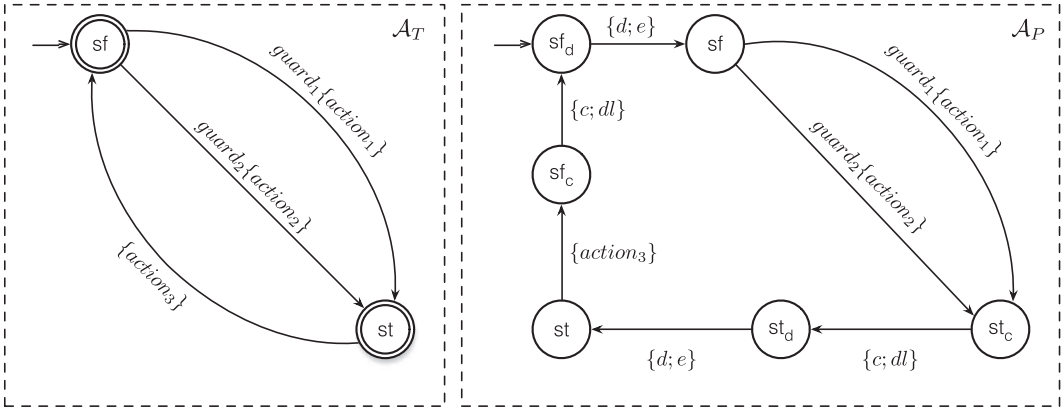

Fig. 4. Graphical representation of a 2-state behavior annex specification to FIACRE*.

Automaton translation In Fig. 4, we give a graphical example of the translation of the state/transition automaton to FIACRE*. Note that the translation is almost direct as the FIACRE* language offers syntactic constructs that are close to the ones defined in the BA. At the process level, we have to deal explicitly with the thread execution model, therefore the thread automaton $\mathcal{A}_T$ is embedded within a larger automaton $\mathcal{A}_P$ that specifies the interactions with the communication ports d, e, c and dl. Initial states of $\mathcal{A}_T$ are moved to the *dispatch* states (sf_d and st_d) in $\mathcal{A}_P$ and terminal states are moved to *complete* states (sf_c and st_c) in $\mathcal{A}_P$. Finally, the generation of the FIACRE* automaton is completed by translating guards and actions of each transition.

2.2.3. Main component generation

The final step of the translation consists in generating a FIACRE* Component modeling the AADL System instance.

Notations As mentioned before, the translation takes as input an AADL model and produces an FIACRE* source file. In an attempt to give a somewhat formal presentation we introduce some notations.

Text written with a sans-serif font refers to the target language, namely FIACRE*. Expressions enclosed in brackets [ ], called *printers* are expanded as FIACRE* source code as part of the translation process. To summarize, any string expression s may be used as a printer $[s]$. In the following, we present some extensions on printers in order to work with collection of values.

Given an iterable collection $C = \{c_1, \ldots, c_n\}$ and a C-indexed family of functions f, also denoted as $[\ f_c \mid c \in C\ ]$ to make explicit its index set, we write $[\ f_c \mid c \in C\ ]_\star$ for the printer which expands each element $c \in C$ with f separated by $\star$. For example, the expression below is expanded as follows:

$$[\ f_c \mid c \in C\ ]_\star \rightsquigarrow \left[\ f_{\sigma(c_1)}\ \right] \star \left[\ f_{\sigma(c_2)}\ \right] \star \cdots \star \left[\ f_{\sigma(c_n)}\ \right]$$

where $C = \{c_1, \ldots, c_n\}$ and $\sigma \in S_n$ is an internally chosen permutation of $\{1, \ldots, n\}$. As a result, the order of expansion is not specified.

When the collection C is ordered over a relation $\preceq$, written $C_\preceq$, the expression below is expanded as:

$$\left[\ f_c \mid c \in C_\preceq\ \right]_\star \rightsquigarrow \left[\ f_{c_1}\ \right] \star \left[\ f_{c_2}\ \right] \star \cdots \star \left[\ f_{c_n}\ \right]$$

given that $c_1 \preceq c_2 \preceq \cdots \preceq c_n$.

In addition, we define the $\oplus$ operator as:

$$[\ f_a \mid a \in A\ ] \oplus [\ g_b \mid b \in B\ ] \triangleq \left[\ x \mapsto \begin{cases} f_x & \text{if } x \in A \\ g_x & \text{if } x \in B \end{cases} \mid x \in A \uplus B\ \right]$$

Finally, we note $\mathcal{T}$ for the set of threads in an AADL system, and given a thread t we note $\mathcal{P}_t$ the set of its ports and $\mathcal{D}_t$ its data access features.

Variable declaration section Each port declared on a thread is associated with a variable in order to desynchronize port communication (sender/receiver) to conform to AADL semantics. Variables are also used to implement data access. Thus, for each data subcomponent that provides a data access we associate a variable. Finally, all these variables are shared between processes and controllers.

Port declaration section Ports are declared in a similar fashion as variables. The main difference lies in the fact that timing constraints can be associated to each port. For example, the port declaration $p : \tau$ in $[a, b]$ specifies that p has type τ with a timing constraint of $[a, b]$. In the context of this translation, timing constraints are used to model the Period and Dispatch properties associated to threads, as described in Fig. 5.

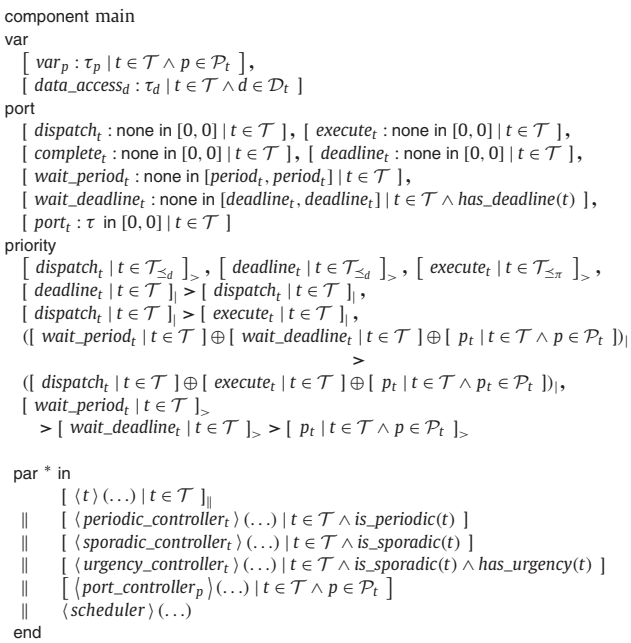

Fig. 5. FIACRE* component definition.

Port priorities section In addition to timing constraints, priorities can be specified on port-based communications. Fig. 5 contains an example of such specification. Some priorities are specified to conform to the semantics of the AADL runtime, for instance, dispatch events are meant to be before execute events and as such, are declared with a higher priority. We write $[\ p \mid p \in P_{\preceq_p}\ ]$ for an *arbitrary* total order $\preceq_p$ among ports in the set P. Such arbitrary total orders are used to prevent state space explosion for model-checking based verification. A special total order $\preceq_\pi$ is used to order ports according to the thread priority property (SEI::Priority). Finally, we write $[\ p \mid p \in P\ ]_| > [\ q \mid q \in Q\ ]_|$ to specify that any port in P has a greater priority than any port in Q.

Port controllers Each port declared at the interface level of an AADL thread is associated with a port controller in FIACRE*. Different controllers are available depending on the properties attached to a port declaration. The properties considered are:

- `Urgency`
- `Queue size`
- `Overflow Handling Protocol`
- `Immediate / Delayed`

The `Urgency` controller is somewhat special because it is not available in FIACRE* RT library and must be generated for each *sporadic* thread with incoming event (data) port specifying an urgency property. To this end, we use a special construct available in FIACRE* which allows to express an order among non-deterministic choices. For example, the following FIACRE* statement that uses this construct

```
select
  transition_1 [] transition_2
unless
  transition_3
end
```

specifies that a transition in the group (`transition_1` and `transition_2`) is possible *only if* the transition `transition_3` is not possible. As a result, it suffices to partition the set of incoming event (data) port according to the `Urgency` property value and use the `select ... unless ... end` statement to force transitions in the intended order.

Parallel composition section The last step of the translation consists in computing the parallel composition of processes. This composition is expressed with

$$\text{par}\Sigma \text{ in } p_1 \| \cdots \| p_n \text{ end}$$

where Σ represents the set of synchronizing ports and $p_1, \ldots, p_n$ are processes.

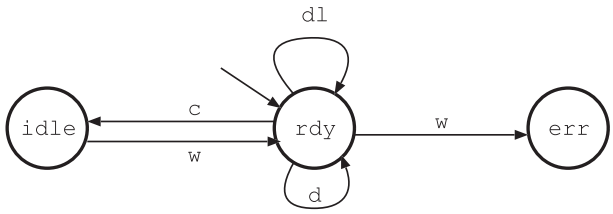

Fig. 6. A simplified view of the periodic controller.

In Fig. 5, we use $\Sigma = *$ to indicate that we want to perform a synchronous parallel composition over all ports declared in the Component.

Finally, processes obtained from AADL threads through translation are instantiated along with their port and shared variable arguments. Each process is associated with its corresponding process controller: `periodic_controller` for periodic threads and `sporadic_controller` for sporadic threads. The FIACRE* code for the `periodic_controller` is given at Section 2.3.1. Then, the connections between port, shared variables and port properties are performed by instantiating the corresponding `port_controller`. Again, an example of such controller is given at Section 2.3.3. Finally, a scheduler process is instantiated.

2.3. FIACRE* *RT library*

The *main* FIACRE* component produced by the translation depends on library components which define protocols introduced by the AADL runtime. They are often generic and use the FIACRE* extensions. These components can be classified as follows:

- Thread dispatch controllers: they aim at sending a dispatch event to applicative thread. This event is sent periodically for periodic threads or after an event has been received and the minimal period has elapsed for a sporadic thread. The controller checks also that completion occurs in time. Otherwise, an error is signaled.
- Schedulers: these components grant processor access to dispatched threads, depending on the scheduling protocol.
- Input ports buffers: these components store input messages and deliver them to the thread at the time defined by the protocol (at dispatch, on demand, ...). Input overflows and freshness of message are detected and managed by these components.
- Connectors: these components transmit data from output buffers to input buffers at the time specified by the protocol.

In the following, we consider one representative for each family of protocol.

2.3.1. *Thread dispatch controllers*

The dispatch controller is synchronized with the controlled applicative thread through its *dispatch* (d), *completion* (c) and *deadline* (dl) ports. An additional port (w) is used as a timer and allows a synchronization on it at each period. The behavior of the controller is illustrated by Fig. 6. Its interface is the following:

```
process periodic_controller[d:none, c:none, dl:in none, w:none]
```

- In the *rdy* state, the controller waits for the thread to synchronize on the *dispatch* port *d*. This synchronization is supposed to be possible without delay. Then, the controller waits for either the completion of the thread (port *c*) or for the end of the period (port *w*). In the former case, the controller transits to the *idle* state. In the latter case, an error is detected (deadline is not respected).
- In the *idle* state, the controller waits for the end of the period (on port *w*), which is also here the instant of the deadline and of the next dispatch. It then transits back to the *rdy* state and sends a deadline event (port *dl*) to trigger data transfers.

It is important to note that the correctness of this model depends on the following points:

- The environment (the applicative thread) accepts the events *d*, *c* and *dl* repeatedly and in this order.
- The environment does not delay the synchronization on port *d*.
- The *w* port is not delayed by the environment and is timed $[T, T]$, where T is both the period and the deadline of the thread.

2.3.2. *Schedulers*

The scheduler interacts with the applicative threads through the ports *execute* and *complete*. A synchronization on the former port means a request for the processor resource. A synchronization on the latter port means the resource is released.

The following code models a very simple scheduler which only guarantees that the processor is allocated to at most one thread. The allocator is generic, parameterized by the number of client threads. Each thread communicates with the scheduler through a private port. A priority can be associated to threads through the use of priorities on FIACRE ports. In FIACRE, an enabled transition on a port prevents execution of any transitions on lower priority ports.

```
process scheduler<|N|>[e: array N of in out none,
                       c: array N of in out none] is
states free, busy
var x: 0..N-1 := 0
init to free
from free
  select i of [N]
    e[i]; x := i
  end; to busy
from busy c[x]; to free
```

2.3.3. Input port buffers

Input port buffers store incoming messages and deliver them to the thread at the time defined by the AADL semantics. Several library elements or parameters adapt the behavior of the port with respect to the kind of the ports (event, data, event data); the scheduling protocol of the attached thread; the management of overflows or the queuing protocol. As an example, we consider an input event port attached to a periodic process. It is supposed to have a capacity of N events which are delivered one at a time, and to block on overflow in order to make it easy to check. The event is received through a synchronization on port e. The event counter is then incremented. When the attached thread is dispatched, one event is transmitted, if available, through the shared variable referenced by ip. Thus, ip receives either the value 0, or the value 1, in which case the counter is decremented.

```
process per_ieport_one_ovf<|N|>[e, d:none](&ip:0..1) is
states s0, ovf
var evt: 0..N := 0
from s0
  select
     e; on evt < N; evt := evt + 1; to s0
  [] e; on evt = N; to ovf
  [] ip := evt > 0?1:0; d; evt := (evt>0?evt - 1:evt); to s0
  end
```

2.3.4. Connectors

Connectors transmit messages from output ports to input ports. Several library elements or parameters are used to select the desired protocol. As an example, we consider a generic process implementing a delayed connection. It is parameterized by the type T of transferred data. It takes as parameter two ports synchronized respectively to the deadline of the emitting thread and to the dispatch of the receiving thread, and references to shared variables containing respectively at most one incoming message (in ip), the outgoing message (in o_v) and its freshness (in o_f). On emitter deadline, a fresh message is taken from the queue (if non-empty) and stored locally. On receiver dispatch, the stored message and its freshness are transmitted. The stored message becomes not fresh.

```
process delayed_connection<|T|>
  [dl_o,d_i: none](&ip: queue 1 of T, &o_v:T, &o_f: bool) is
states s0
var x: T, f: bool := false
init to s0
from s0
  select
     dl_o; on not (empty(ip));
         x := first(ip); ip := {||}; f := true; to s0
  [] dl_o; on empty(ip); to s0
  [] o_v := x; o_f := f; d_i; f := false; to s0
  end
```

2.4. Prototype implementation

Following the translation procedure presented in the previous section, we realized a prototype implementation of a translator from an AADL model to FIACRE. In the rest of this section, we present the tools we used and give an overview of the toolchain.

The translation from AADL to FIACRE is a two-steps process (Fig. 7). We focus our attention to the first step. The development environment we used to implement our translator is based on the Eclipse platform and particularly the Eclipse Modeling Tools.

Fig. 7. Toolchain overview of the translator from AADL to FIACRE.

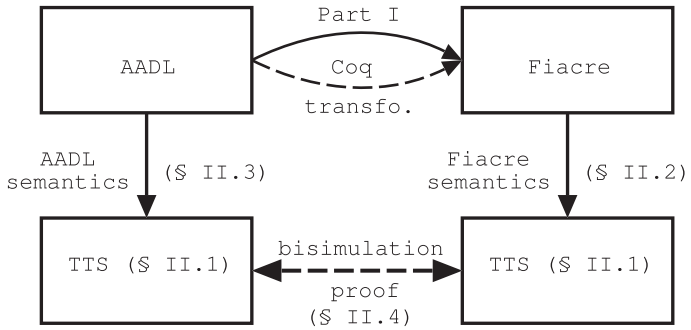

Fig. 8. Part 2 map.

One of the main plug-in we used is *Osate* (version 1.5.8) which is an open-source plug-in that provides a set of tools based on the Eclipse Modeling Framework (EMF). Osate gives access to a set of services such as: a parser for the textual AADL representation, tools to analyze AADL model, semantics checker, and it offers the possibility to extract a model and instance model of a system described as a textual model in AADL language. This model and instance model representation is described in file format called XMI (XML Metadata Interchange) which is the starting point of our transformation to FIACRE*.

Another tool we used is a plug-in called *Acceleo* [17] (version 3.0) which is an implementation of the Model to Text (M2T) as defined by the Object Management Group (OMG). From a metamodel and a model (represented in XMI format) we can use Acceleo to produce text-based output, in our case an FIACRE* file. Model to Text is based around two fundamental primitives: *templates* and *queries*. In short, templates are used to produce a textual output whereas queries are used to extract information out of the model in a declarative manner. Finally, the translation from FIACRE* to FIACRE is done with JAVA and TOM [29]. TOM is a rewriting engine which provides syntax extensions to JAVA to define transformations.

The combination of these tools allows us to obtain a working prototype fully integrated within the Eclipse platform. We have experimented the toolchain on some case studies. As an example, we present in the companion paper [44] the APOTA case study which is a file transfer protocol used in the avionics domain. The model checking results can be found in [42]. They illustrate the verification of some LTL properties superposed to the generated FIACRE code.

In addition, the high-level constructs provided by FIACRE* along with its RT library have considerably eased the process of translating AADL to FIACRE.

Part 2. On the correctness of the AADL to FIACRE transformation

The objective of this part is to give hints on how the toolchain presented in Part 1 could be verified. Verifying such a toolset is a long term goal. We mainly investigate here one element of the toolchain which is the transformation from AADL to FIACRE. Verifying such a transformation means ensuring that the truth or falsity of properties expressed on an AADL model is preserved by the transformation. We consider linear temporal logic properties and bi-simulation between behavioral models of the source and target models, which is known to preserve this class of properties. In order to compare the behavior of the source AADL model and of the resulting FIACRE model, we introduce a common framework: *timed transition systems* [23] (TTS for short, see Definition 6). Then, we express the semantics of an abstract view of FIACRE and of a *reduced* subset of AADL in terms of timed transition systems. Lastly, we discuss about the verification of the transformation. These steps are represented by Fig. 8 where dashed lines are left for future work.

3. Common semantics domain

The present section recalls the basis of transition systems theory, and extends classical Labeled Transition Systems [4] by distinguishing global labels from local ones. Indeed, global labels are needed to specify communication or synchronization with other components while local labels are used to specify internal actions. The set of all labels of a transition system is $L \stackrel{def}{=} L_{\mathcal{G}} \uplus L_{\mathcal{L}}$, where $L_{\mathcal{L}}$ is the set of local labels (or tau) of the system.

Also, we distinguish *shared memory* from *private*. We assume that all the transition systems share a set of global states $S_{\mathcal{G}}$ and have each their own set of local states $S_{\mathcal{L}}$. Shared memory allows processes to exchange data and communicate without need to synchronize and use ports. In some cases it may be very convenient to use shared memory as long as two processes never write at the same location at the time. This is especially true for shared memory architectures.

Our formalization of the memory of transition systems considers that local states are dependent to global states. Thus, we split $S_{\mathcal{L}}$ with respect to $S_{\mathcal{G}}$ such that $S_{\mathcal{L}} = \bigcup_{g \in S_{\mathcal{G}}} S^g_{\mathcal{L}}$. The set of states of a given transition system (mixing shared and private memory) is of the form of $S \stackrel{def}{=} \{(g, l) \mid g \in S_{\mathcal{G}} \wedge l \in S^g_{\mathcal{L}}\}$.

Then, because of our desire to define a compositional model, we introduce a partial function $\mathsf{mrg} : S_{\mathcal{G}} \times S_{\mathcal{G}} \to S_{\mathcal{G}}$ that computes the merge of two shared states. This function cannot be defined here because $S_{\mathcal{G}}$ is not structured (by set of shared variables for example) and we merely specify it. Thereby, we consider that mrg should be *commutative, associative* and *idempotent*. Formally it means that whatever the definition of mrg, it must satisfy the following constraints when terms are defined:

- $\forall g_1, g_2 \in S_{\mathcal{G}}, \mathsf{mrg}(g_1, g_2) = \mathsf{mrg}(g_2, g_1)$
- $\forall g_1, g_2, g_3 \in S_{\mathcal{G}}, \mathsf{mrg}(g_1, \mathsf{mrg}(g_2, g_3)) = \mathsf{mrg}(\mathsf{mrg}(g_1, g_2), g_3)$
- $\forall g \in S_{\mathcal{G}}, \mathsf{mrg}(g, g) = g$

As an example, we can define $S_{\mathcal{G}}$ as the set of partial valuations of a set of global variables $X_{\mathcal{G}}$ in a domain D: $S_{\mathcal{G}} \stackrel{def}{=} X_{\mathcal{G}} \nrightarrow D$. Then $\mathsf{mrg}(g_1, g_2)$ will be the overloading function when g_1 and g_2 are compatible, i.e. when they are both defined on a variable, they assign it the same value.

3.1. Labeled transition systems

The classical model of *labeled transition systems* [4] (LTS for short) is usually used to give a mathematical representation to programs and more recently to component-based systems. We introduce now this model in a *shared/private* memory context.

Definition 1 *(Labeled transition systems).* An LTS defined over $L_{\mathcal{G}}$ and $S_{\mathcal{G}}$ (introduced above) is a 4-tuple $lts \stackrel{def}{=} \langle L_{\mathcal{L}}, S_{\mathcal{L}}, \boldsymbol{init}, \boldsymbol{next}\rangle$, where:

- $L_{\mathcal{L}}$ is the set of *local labels* of *lts*.
- $S_{\mathcal{L}} (= \bigcup_{g \in S_{\mathcal{G}}} S^g_{\mathcal{L}})$ is a set of *local states* (or stores) and we define S as the set $\{(g, l) \mid g \in S_{\mathcal{G}} \wedge l \in S^g_{\mathcal{L}}\}$.
- $\boldsymbol{init}$ is a predicate over S that defines the initial states of *lts*.
- $\boldsymbol{next}$ is a predicate over $S \times L \times S$; $\boldsymbol{next}$ defines the set of transitions of *lts* that are triplets of the form (s, ℓ, s'), where s is the source state, ℓ is the taken label and s' the target state of the transition.

From now, in this general setting, we consider that all the LTS are defined over some given sets $L_{\mathcal{G}}$ and $S_{\mathcal{G}}$.

Definition 2 *(Enabled labels).* Assuming an LTS $\langle L_{\mathcal{L}}, S_{\mathcal{L}}, \boldsymbol{init}, \boldsymbol{next}\rangle$, a label $\ell \in L$ is *enabled* from a state $s \in S$, if there is a state $s' \in S$ such that the triplet (s, ℓ, s') belongs to $\boldsymbol{next}$. Formally, we define the predicate $\boldsymbol{enabled}$ over $S \times L$ as $\boldsymbol{enabled}_s(\ell) \stackrel{def}{=} \exists s' \in S, \boldsymbol{next}(s, \ell, s')$.

Definition 3 *(Simulation relations on LTS).* Given two LTSs, namely $lts^\flat$ (the concrete) and $lts^\natural$ (the abstract), defined as $\langle L^i_{\mathcal{L}}, S^i_{\mathcal{L}}, \boldsymbol{init}_i, \boldsymbol{next}_i\rangle$ for $i \in \{\flat, \natural\}$, $lts^\flat$ is simulated by $lts^\natural$ through relations $R_S \subseteq S^\flat \times S^\natural$ and $R_L \subseteq (L^\flat \times L^\natural)$ if:

$$
\begin{array}{l}
\quad \forall s_\flat \in S^\flat, \boldsymbol{init}_\flat(s_\flat) \Rightarrow \exists s_\natural, \boldsymbol{init}_\natural(s_\natural) \wedge (s_\flat, s_\natural) \in R_S \\
\bigwedge \\
\quad \forall s_\flat s'_\flat \in S^\flat, \forall s_\natural \in S^\natural, \forall \ell_\flat \in L_{\mathcal{G}} \cup L^\flat_{\mathcal{L}}, \\
\qquad (\boldsymbol{next}_\flat(s_\flat, \ell_\flat, s'_\flat) \wedge (s_\flat, s_\natural) \in R_S) \Rightarrow \\
\qquad\quad (\exists s'_\natural \in S^\natural, \exists \ell_\natural \in L_{\mathcal{G}} \cup L^\natural_{\mathcal{L}}, \boldsymbol{next}_\natural(s_\natural, \ell_\natural, s'_\natural) \wedge \\
\qquad\qquad\quad (s'_\flat, s'_\natural) \in R_S \wedge (\ell_\flat, \ell_\natural) \in R_L).
\end{array}
$$

We formally write it by $lts^\flat \sqsubseteq_{(R_S, R_L)} lts^\natural$.

Definition 4 *(Bisimilar LTSs).* Given two LTSs, namely $lts^\flat$ and $lts^\natural$, defined as $\langle L^i_{\mathcal{L}}, S^i_{\mathcal{L}}, \boldsymbol{init}_i, \boldsymbol{next}_i\rangle$ for $i \in \{\flat, \natural\}$, $lts^\flat$ is bisimilar to $lts^\natural$ if there exists two relations $R_S \subseteq S^\flat \times S^\natural$ and $R_L \subseteq L^\flat \times L^\natural$, such as:

$$lts^\flat \sqsubseteq_{(R_S, R_L)} lts^\natural \ \wedge \ lts^\natural \sqsubseteq_{(R_S^{-1}, R_L^{-1})} lts^\flat$$

We formally write it by $lts^\flat \simeq lts^\natural$.

3.2. Composition of LTSs

Reasoning about concurrent systems requires interpretation of process composition in the chosen semantic domain. For such a purpose, we define a binary synchronous product of LTSs.

Definition 5 *(Composition of LTSs).* The *composition* of two LTSs, namely lts_1 and lts_2, defined as $\langle L^i_{\mathcal{L}}, S^i_{\mathcal{L}}, \boldsymbol{init}_i, \boldsymbol{next}_i\rangle$ for $i \in \{1,2\}$, over a set of synchronizable labels $L_{\mathcal{S}} \subseteq L_{\mathcal{G}}$, is an LTS, $(lts_1 \parallel_{L_{\mathcal{S}}} lts_2) \stackrel{def}{=} \langle L_{\mathcal{L}}, S_{\mathcal{L}}, \boldsymbol{init}, \boldsymbol{next}\rangle$, where:

- $L_{\mathcal{L}} \stackrel{def}{=} L^1_{\mathcal{L}} \uplus L^2_{\mathcal{L}}$
- $S_{\mathcal{L}}$ is defined as the set $S^1_{\mathcal{L}} \times S^2_{\mathcal{L}}$
- $\boldsymbol{init}(\langle g, (l_1, l_2)\rangle) \stackrel{def}{=} \boldsymbol{init}_1(\langle g, l_1\rangle) \wedge \boldsymbol{init}_2(\langle g, l_2\rangle)$
- $\boldsymbol{next}(\langle g, (l_1, l_2)\rangle, \ell, \langle g', (l'_1, l'_2)\rangle) \stackrel{def}{=}$

$$\bigvee \left(\begin{array}{l} (1)\ \ell \in L^1_{\mathcal{L}} \ \wedge\ \boldsymbol{next}_1(\langle g, l_1\rangle, \ell, \langle g', l'_1\rangle) \ \wedge\ l_2 = l'_2 \\ (2)\ \ell \in L^2_{\mathcal{L}} \ \wedge\ \boldsymbol{next}_2(\langle g, l_2\rangle, \ell, \langle g', l'_2\rangle) \ \wedge\ l_1 = l'_1 \\ (3)\ \ell \in L_{\mathcal{G}} \ \wedge\ \ell \notin L_{\mathcal{S}} \ \wedge\ \boldsymbol{next}_1(\langle g, l_1\rangle, \ell, \langle g', l'_1\rangle) \ \wedge\ l_2 = l'_2 \\ (4)\ \ell \in L_{\mathcal{G}} \ \wedge\ \ell \notin L_{\mathcal{S}} \ \wedge\ \boldsymbol{next}_2(\langle g, l_2\rangle, \ell, \langle g', l'_2\rangle) \ \wedge\ l_1 = l'_1 \\ (5)\ \ell \in L_{\mathcal{S}} \ \wedge\ \exists g'_1 g'_2 \in S_{\mathcal{G}}, \bigwedge \left(\begin{array}{l} g' = \mathsf{mrg}(g'_1, g'_2) \\ \boldsymbol{next}_1(\langle g, l_1\rangle, \ell, \langle g'_1, l'_1\rangle) \\ \boldsymbol{next}_2(\langle g, l_2\rangle, \ell, \langle g'_2, l'_2\rangle) \end{array} \right) \end{array} \right)$$

3.3. Timed transition systems

We recall now the definition of *Timed Transition Systems* which is commonly used to define the semantics of component-based system which embeds real-time features as AADL architectures do.

Definition 6 *(Timed transition systems).* A *Timed Transition System* (TTS for short) is an LTS, $\langle L_{\mathcal{L}}, S_{\mathcal{L}}, \boldsymbol{init}, \boldsymbol{t\_next}\rangle$, defined over $L_{\mathcal{G}} \cup \mathbb{R}^+$ and $S_{\mathcal{G}}$. Thus, there are two kinds of transition relations: *discrete* and *delay*. Delay transitions are required to obey the following properties:

- zero delay: $\forall \mathsf{sto} \in S, \mathsf{sto} \xrightarrow{0} \mathsf{sto}$
- determinism: $\forall \mathsf{sto}, \mathsf{sto}', \mathsf{sto}'' \in S, \forall \delta \in \mathbb{R}^+,$

$$\mathsf{sto} \xrightarrow{\delta} \mathsf{sto}' \wedge \mathsf{sto} \xrightarrow{\delta} \mathsf{sto}'' \Rightarrow \mathsf{sto}' = \mathsf{sto}''$$

- additivity: $\forall \mathsf{sto}, \mathsf{sto}', \mathsf{sto}'' \in S, \forall \delta, \delta' \in \mathbb{R}^+,$

$$\mathsf{sto} \xrightarrow{\delta} \mathsf{sto}' \wedge \mathsf{sto}' \xrightarrow{\delta'} \mathsf{sto}'' \Rightarrow \mathsf{sto} \xrightarrow{\delta+\delta'} \mathsf{sto}''$$

- continuity: $\forall \mathsf{sto}, \mathsf{sto}'' \in S, \forall \delta', \delta'' \in \mathbb{R}^+,$

$$\mathsf{sto} \xrightarrow{\delta'+\delta''} \mathsf{sto}'' \Rightarrow \exists \mathsf{sto}', \mathsf{sto} \xrightarrow{\delta'} \mathsf{sto}' \wedge \mathsf{sto}' \xrightarrow{\delta''} \mathsf{sto}''$$

We note $(\mathsf{sto} \xrightarrow{\delta} \mathsf{sto}')$ for $\boldsymbol{t\_next}(\mathsf{sto}, \delta, \mathsf{sto}')$.

3.4. Composition of TTSs

We consider the composition operation on TTSs, which is used to define the semantics of an AADL model as the composition of the semantics of its constituents.

Definition 7 *(Composition of TTSs).* The *composition* of two TTSs, namely tts_1 and tts_2, defined as $\langle L^i_{\mathcal{L}}, S^i_{\mathcal{L}}, \boldsymbol{init}_i, \boldsymbol{t\_next}_i\rangle$, for $i \in \{1,2\}$, over a set of synchronizable ports $L_{\mathcal{S}} \subseteq L_{\mathcal{G}}$, is a TTS defined as a composed LTS, $(tts_1 \parallel_{L_{\mathcal{S}}} tts_2) \stackrel{def}{=} \langle L_{\mathcal{L}}, S_{\mathcal{L}}, \boldsymbol{init}, \boldsymbol{t\_next}\rangle$, but where $\boldsymbol{t\_next}$ is a predicate over $\mathcal{S} \times (L \cup \mathbb{R}^+) \times \mathcal{S}$ defined as:

$$\diamond\ \textbf{\textit{t\_next}}(\langle g,(l_1,l_2)\rangle,\ell,\langle g',(l'_1,l'_2)\rangle) \stackrel{def}{=}$$

$$\bigvee \left(\begin{array}{l} (1)\ \ell \in L^1_{\mathcal{L}} \ \wedge\ \textbf{\textit{t\_next}}_1(\langle g,l_1\rangle,\ell,\langle g',l'_1\rangle)\ \wedge\ l_2 = l'_2 \\ (2)\ \ell \in L^2_{\mathcal{L}} \ \wedge\ \textbf{\textit{t\_next}}_2(\langle g,l_2\rangle,\ell,\langle g',l'_2\rangle)\ \wedge\ l_1 = l'_1 \\ (3)\ \ell \in L_{\mathcal{G}} \ \wedge \ell \notin L_{\mathcal{S}} \ \wedge\ \textbf{\textit{t\_next}}_1(\langle g,l_1\rangle,\ell,\langle g',l'_1\rangle)\ \wedge\ l_2 = l'_2 \\ (4)\ \ell \in L_{\mathcal{G}} \ \wedge \ell \notin L_{\mathcal{S}} \ \wedge\ \textbf{\textit{t\_next}}_2(\langle g,l_2\rangle,\ell,\langle g',l'_2\rangle)\ \wedge\ l_1 = l'_1 \\ (5)\ \ell \in L_{\mathcal{S}} \ \wedge\ \exists g'_1 g'_2 \in S_{\mathcal{G}}, \bigwedge \left(\begin{array}{l} g' = \mathsf{mrg}(g'_1,g'_2) \\ \textbf{\textit{t\_next}}_1(\langle g,l_1\rangle,\ell,\langle g'_1,l'_1\rangle) \\ \textbf{\textit{t\_next}}_2(\langle g,l_2\rangle,\ell,\langle g'_2,l'_2\rangle) \end{array} \right) \end{array} \right)$$

$$\diamond\ \textbf{\textit{t\_next}}(\langle g,(l_1,l_2)\rangle,\delta,\langle g',(l'_1,l'_2)\rangle) \stackrel{def}{=}$$

$$\delta \in \mathbb{R}^+ \ \wedge\ \exists g'_1 g'_2 \in S_{\mathcal{G}}, \bigwedge \left(\begin{array}{l} g' = \mathsf{mrg}(g'_1,g'_2) \\ \textbf{\textit{t\_next}}_1(\langle g,l_1\rangle,\delta,\langle g'_1,l'_1\rangle) \\ \textbf{\textit{t\_next}}_2(\langle g,l_2\rangle,\delta,\langle g'_2,l'_2\rangle) \end{array} \right)$$

In other words, we treat discrete transitions and delay transitions separately.

4. FIACRE kernel mechanization

Following the principle of the *pivot language* FIACRE, we have introduced a *semantic model midway* between FIACRE and TTS. Indeed, this model describes priority and time features as syntactic constraints (while time is specified dynamically in TTSs) but on the other hand it specifies systems in mathematical terms (while it is programming in FIACRE).

The purpose is to give a *mechanized semantics* to the kernel of FIACRE in a proof-assistant in order to be able to reason formally about FIACRE systems and prove the transformation from AADL to FIACRE. We present in the following of this section this semantic model as close as possible of its mechanization in the COQ proof assistant.

4.1. Time constrained transition systems

As for the LTS (see Definition 3.1), we distinguish global *labels* (or events) from local ones here. But here, we introduce a *finite set* of ports under the *infinite set* of labels. Among others, this distinction is needed to go from *open systems* which can communicate or synchronize with others to *closed systems* that only have internal (or local) actions. Doing so requires to *hide* the set of global events but since this set may be infinite, we handle this task through their corresponding ports which are finite in number. Thus, in the following we assume that any transition system is defined over a finite set of global ports $P_{\mathcal{G}}$ which induces an implicit set of global labels $L_{\mathcal{G}}$. We assume also that the set of all ports of any transition system is of the form $P \stackrel{def}{=} P_{\mathcal{G}} \uplus P_{\mathcal{L}}$ where $P_{\mathcal{L}}$ is the set of local ports of the given transition system (see Definition 8).

Moreover, introducing *priorities* can be useful to control, preserve and try to guarantee, for example, the deadlock-freedom of such systems [19]. Thus, we provide to our Transition System based model (Definition 8) a priority relation, which is a strict partial order over ports so that only labels linked to ports with maximal priority can be fired at some point of the execution. The meaning of a *priority relation* over two ports p and p' of a transition system, is that if p has priority over p' then every transition through p' cannot be taken if a label through p is enabled. We formalize in our framework a priority relation, $\prec$, as an irreflexive and transitive relation. Consequently, a priority relation is also *acyclic*. An important property over priority relations that our transition systems must satisfy, is that *ports* that have priority over others *must be local*. Without this constraint, our semantic model is no longer compositional.

Then, we have associated *time interval constraints* to the ports of the model, in the same way as T. Henzinger et al. [22] for labels. Moreover, we add the so-called *reset relation* to this model that enables us to specify which clocks (or timers) are reset after the firing of a given transition. That reset relation is helpful to model directly the semantic differences between the two constructs of FIACRE that allow to pass transitions (i.e. `loop` and `to`, see the periodic controller of Part 1, Section 2.3.1, for example). We name this model *Time Constrained Transition Systems* (TCTS for short) since the term TTS is overloaded and already used here and because time features are only expressed as syntactic timed constraints on transitions.

Definition 8. A Time Constrained Transition System, namely *tcts*, is a 8-tuple defined over a set of values V, a finite set of global ports $P_{\mathcal{G}}$ and a set of states over shared variables $S_{\mathcal{G}}$, $\langle P_{\mathcal{L}}, T, S_{\mathcal{L}}, \mathsf{val}, \mathsf{prt}, \textbf{\textit{init}}, \textbf{\textit{next}}, \mathcal{R}, I \prec\rangle$, where:

- $P_{\mathcal{L}}$ is the (finite) set of *ports* of *tcts*. We define $P = P_{\mathcal{L}} \cup P_{\mathcal{G}}$.
- T is the set of *(the names of) transitions* of *tcts*.
- $S_{\mathcal{L}}$ is the set of *states* (or stores) of *tcts*.
- prt is a function from T to P, that associates to every transition a unique *port*.
- val is a function from $T \times S$ to V, that associates for all transition, in every state, a *value*. It defines, together with a port of the considered transition, the *label* of the transition. Thus, the set L of labels is defined here by $P \times V$.

```
process Pattern [p : T](& v : T) is
  ports p' in [m,M]
  var x : T
  states s, init s

    from s select
      []_i^n tr_i
    end
```

where `[]` is the parallel operator of Fiacre and for all $i \leq n$ (n is the number of concurrent transitions from `s`), tr_i has one of the four following patterns:

- T_1 : `on g; stm1; p!e; stm2; to s`
- T_2 : `on g; stm1; p!e; stm2; to loop`
- T_3 : `on g; p'; stm; to s`
- T_4 : `on g; p'; stm; to loop`

Fig. 9. Patterns of Fiacre transitions illustrated by a minimal process.

- ***init*** is a non-empty subset of S that defines the initial states of *tcts*.
- ***next*** defines the set of transitions of *tcts* that are triplets of the form $(\text{sto}, tr, \text{sto}') \in S \times T \times S$, also denoted as $\text{sto} \xrightarrow{tr} \text{sto}'$, where $\text{sto} \in S$ is the source state, $tr \in T$ is the name of the taken transition and $\text{sto}' \in S$ the target state of the transition.
- $\mathcal{R}$ is the reset transition relation. $(tr, tr') \in \mathcal{R}$ states that at *execution*, the firing of tr resets the *implicit clock* of tr'. Otherwise, the implicit clock associated to tr' keeps running. For all $tr \in T$, $(tr, tr) \in \mathcal{R}$ ($\mathcal{R}$ is reflexive).
- I is a function that assigns to every port $p \in P$ a non-empty interval of $\mathbb{R}^+$. I_p (or $I(p)$) specifies both minimal (lower) and maximal delay (upper bound) to elapse once a transition through p has been enabled
- $\prec$ is a priority relation over P. The meaning of the priority relation over two ports p and p' of a transition system, is that if p has priority over p' then every transition through p' cannot be taken if an other transition through p is enabled.

4.2. Semantics of Fiacre transitions in terms of TCTS

We illustrate in this section what is a *standard* Fiacre *transition* and what is its semantics in the TCTS representation. Thus, a Fiacre transition is defined following one of the four patterns of the process defined on Fig. 9, where `s` is a location, `g` (the guard) is a boolean expression, `p` and `p'` are ports, `stm`, `stm1` and `stm2` (the statements) are sequences of imperative instructions. `stm1` will be executed before the synchronization on `p` and `stm2` thereafter. In order to avoid conflicts, we assume some restrictions about `stm1` and `stm2`: Both has to be defined only over local variables (only `x` in our example) of the process while `stm` can assign shared variables (only `v` in our example) of the whole system. Also, `e` is an expression of type `T` that is used to encode the data exchanged with another process.

In transitions of type T_1 or T_2, `p` is a global port, used for synchronization or the exchange of data between processes, while in T_3 and T_4 `p'` is local, only used in order to delayed the action of the transitions. A time *could* be associated to `p` in the environment (the component) where this process will be put in interaction with other processes. A time interval interval *has to* be associated to `p'` in the process otherwise local ports are useless since they only serve to delay transition. Here the interval is `[m,M]` meaning in one hand that transitions of type T_3 or T_4 have to be continuously enabled for at least `m` units of time and on the other hand have to be taken before `M` units of time while being continuously enabled for that long.

Regarding to the exchange of data, the standard *reception symbol* `-?-` (as in CSP) is substituted here by a `-!-` and an assignment in advance. For instance, the instruction ⟨`p?x`⟩ is written ⟨`x := any; p!x`⟩ in Fiacre, where ⟨`x := any`⟩ can be interpreted as a non-deterministic assignment to `x`. Nevertheless, only one value assigned to `x` will match with the value of the sender process and synchronization are done if and only if the port and the value (of the given expression) are the same on both side of a communication. For instance, ⟨`p!3`⟩ means that the value 3 has to be observed on the port `p` (but this will have no effect on any variable of the process). Also, ⟨`p!x`⟩ means that the value of `x` has to be observed on both side of the port `p` (but this will also have no effect on any variable of the process, either `x`). However, in this case on one side (of the receiver) this instruction shall be preceded by ⟨`x := any`⟩ and on the other side (of the sender) there will be nothing more.

Furthermore, each of the transitions of the process (the tr_i, where $1 \leq i \leq n$ with $n \in \mathbb{N}$) has its own clock although they are built only on two different ports here. When taken, transitions of type T_1 or T_3 reset the clock of all the transitions which could be taken from `s`, while those of the form T_2 and T_4 only reset their own clock (i.e. they preserve the time already waited by the others in concurrence).

In order to formalize the semantics of this process `Pattern` which define the different forms of Fiacre transitions, we can give a mathematical representation of it with the TCTS $\langle P_{\mathcal{L}}, T, S, \text{val}, \text{prt}, \textbf{\textit{init}}, \textbf{\textit{next}}, \mathcal{R}, I \prec \rangle$, where:

- $P_{\mathcal{G}} = \{p\}$
- $P_{\mathcal{L}} = \{p'\}$
- $T \subseteq T_1 \uplus T_2 \uplus T_3 \uplus T_4$
- $\forall tr \in T, tr \in T_1 \uplus T_2 \Rightarrow \mathsf{prt}(tr) = p \wedge tr \in T_3 \uplus T_4 \Rightarrow \mathsf{prt}(tr) = p'$
- $\{s\} = V_{loc} \subset V$
- $\exists\ loc \in \mathcal{X}, \forall\ \mathsf{sto} \in S, \mathsf{sto}_x = s$, where $\mathcal{X}$ is the set of variables of the system
- $\forall \mathsf{sto}, \mathsf{sto}' \in S,$
  $\bigwedge \begin{array}{l} \forall tr \in T_1 \uplus T_2, \textbf{\textit{next}}(\mathsf{sto}, tr, \mathsf{sto}') \Rightarrow [\![\mathtt{g}]\!](\mathsf{sto}) \wedge [\![\mathtt{stm1\,;\,stm2}]\!](\mathsf{sto}, \mathsf{sto}') \\ \forall tr \in T_3 \uplus T_4, \textbf{\textit{next}}(\mathsf{sto}, tr, \mathsf{sto}') \Rightarrow [\![\mathtt{g}]\!](\mathsf{sto}) \wedge [\![\mathtt{stm}]\!](\mathsf{sto}, \mathsf{sto}') \end{array}$
- $I_p = [0, \infty[$ and $I_{p'} = [m, M]$
- $\forall tr \in T_1 \uplus T_3, \forall tr' \in T_1 \uplus T_2 \uplus T_3 \uplus T_4, (tr, tr') \in \mathcal{R} \wedge (tr', tr') \in \mathcal{R}$

4.3. Composition and semantic interpretation of TCTS

Reasoning about concurrent systems requires to be able to interpret composition of processes in the chosen semantic domain. For such a purpose, we define a binary synchronous product of TCTS based on the composition of LTS (Definition 5) and the union of priority relations. Moreover, we prove with Coq [6] several theorems such as *commutativity*, *associativity*, *monotonicity* of our binary product, in order to establish its *compositionality*.

Finally, we define the semantics of TCTS in terms of TTS, interpreting *time constraints* by *timed transitions* and filtering non-priority transitions. Doing so requires to add in the state of each target TTS a function that associates to every transition an *explicit clock* that is needed to count elapsed time since this transition is enabled. This semantics interpretation is defined in [18]. The purpose, among others, is to be able to reason on TCTS*s* at the TTS level through timed (bi-)simulation relations. For example, proving that two systems are behaviorally equivalent requires indeed to provide a bi-simulation relation between them (see Definition 4).

Conclusion There are several reasons for wanting to mechanize such theories based on transition systems in a proof assistant rather than building a dedicated automatic tool. The most important reason is certainly that proof assistants like Coq [6], Isabelle-hol [27] or pvs [37], provide a generic and very expressive proof environment to certify *computing patterns*, *programming mechanisms* or *language transformations*. We claim also that the results which are encoded and checked with such tools reach probably the currently highest level of confidence in formal software verification. In other words, proofs built and then checked with, for example Coq, avoid to a very high extent fallacious and incomplete arguments that are so easy and so common to find in standard mathematical proofs. Thus, all we have described in this section have been fully formalized[1] in the syntax of Coq.

Moreover, if one wants to prove that the *periodic controller* (described in Fig. 6) is sound, our mechanization of TCTS*s* can be useful. Doing so needs first to be able to express the correctness property of this controller. Thus, we have implemented in Coq a *linear temporal logic* adapted to the systems we consider and a proof methodology very close to the one proposed in [21], where each proof rule is defined and validated by a Coq theorem. With this, we are able to certify in Coq real-time specifications expressed as temporal formula, but by *reasoning syntactically* on TCTS*s*. Some details are given in [18].

5. AADL mechanization

Describing the formal semantics of the whole AADL language is a long-term goal, as well as formally proving the correctness of its translation to another formalism. In order to illustrate the verification methodology we consider a *reduced* subset of the subset of Part 1 limited to synchronous aspects. As shown in Section 1.1.2 of Part 1, a synchronous AADL execution model is obtained by considering logically synchronized periodic threads communicating through data ports. This subset is usually used in safety-critical systems, to guarantee the determinism and predictability of system behaviors. Multi-partitions and multi-processors mechanisms are excluded, and we just consider simple scheduling features: single-processor and non-preemption. We have mechanized this subset in the proof-assistant Coq [6].[2] Here, we give a Coq-independent mathematical notations.

5.1. Abstract syntax of the synchronous subset of AADL

We give the abstract syntax of the synchronous subset which has a flattened view of the AADL model as a set of communicating threads. The execution of the thread is abstracted by a function $ComputeOutput(Inputs : Iports(th) \rightarrow VALUE)$: $Oports(th) \rightarrow VALUE$. Please notice that DURATION, PORT and VALUE are predefined types.

[1] It can be found at the following address http://www.irit.fr/~Manuel.Garnacho/Coq/MAS.
[2] http://www.irit.fr/~Jean-Paul.Bodeveix/COQ/AADL2TASM/V0/.

```
Type Thread:=
{ Iports: set of PORT;
  Oports: set of PORT;
  Period: DURATION;
  ComputeOutput(Inputs:Iports(th) → VALUE): Oports(th) → VALUE;
  BCET: DURATION;
  WCET: DURATION;
  Deadline: DURATION;
}
Type Connection:=
{ SrcThread: Thread; \ * Source Thread * \
  DstThread: Thread; \ * Destination  Thread * \
  SrcPort: PORT; \ * Source Port * \
  DstPort: PORT; \ * Destination  Port * \
  ConnectionType: {Immediate, Delayed};
}
Type Model:=
{ Threads: set of Thread;
  Connections: set of Connection;
}
```

Listing 1: The abstract syntax of the synchronous subset of AADL.

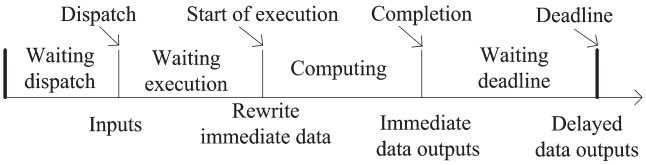

Fig. 10. The time line of an AADL periodic thread with data-port communications.

5.2. Operational semantics of the synchronous subset of AADL

In this section, we first give an informal interpretation of the semantics of the synchronous subset of AADL, and then we define it on the TTSs.

5.2.1. Informal interpretations

Periodic threads A periodic thread is dispatched periodically, and its inputs received from other threads are frozen at dispatch time (by default), i.e., at time zero of the period. As a result the computation performed by a thread is not affected by the arrival of new inputs. Similarly, the outputs are made available to other threads at completion or deadline time, depending on the connection.

Data-port communications between periodic threads First, the communication affects the input/output timing of the periodic threads. (1) For an immediate connection, the execution of the recipient is suspended until the sender completes its execution. As mentioned above, the inputs have been copied at dispatch time, so the recipient needs to replace the old data using the data got from the sender at the start of execution. (2) For a delayed connection, the value from the sender is transmitted at its deadline and is available to the recipient at its next dispatch. The recipient just needs the last dispatch data from the sender. Second, the immediate connection implies a static alignment of thread execution order, i.e., it deals with the scheduling of the sending of messages and the processing of the received messages.

In conclusion, the time line of a periodic thread with data-port communications is represented in Fig. 10.

5.2.2. Formal specification using TTSs

The behavior of a periodic thread (th), including the behavior of its periodic controller illustrated in Fig. 6, is specified as a TTS (called $\text{TTS}_{periodic\_thread}$). The behavior of a data-port connection between periodic threads (cn) is defined as another TTS (called $\text{TTS}_{connection}$). As mentioned above, an AADL model (m) has a set of communicating threads. So, the operational semantics of the synchronous subset of AADL is the composition of a set of TTSs, like the example given in Fig. 11. The definitions of $\text{TTS}_{periodic\_thread}$ and $\text{TTS}_{connection}$ will be given after.

These TTSs communicate through global labels and through a global state. We thus first define the sets $L_{\mathcal{G}}$, $S_{\mathcal{G}}$ and $\boldsymbol{init}(S_{\mathcal{G}})$ and the mrg function which defines how global states updates may be merged during TTS composition.

- $L_{\mathcal{G}} = \{\text{execute}(th), \text{complete}(th), \text{deadline}(th)\}$ is the set of labels used for synchronization between a thread, the scheduler and the connections.
- The global state space is defined by the partial valuations of a set of variables. An *IportBuffer* variable is defined for each input port of each thread. The sender copies values of output ports to the buffer, and the recipient copies values

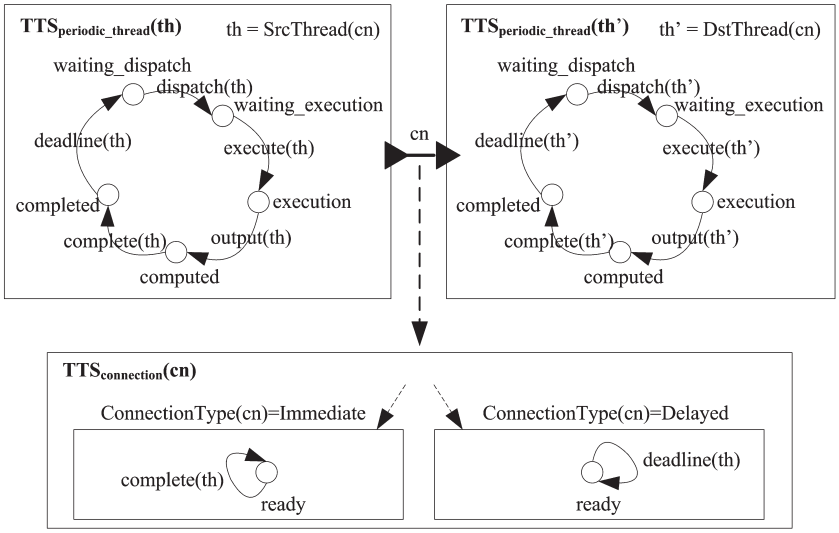

Fig. 11. The operational semantics of an AADL model including two periodic threads and one data-port connection.

from the buffer to the input ports. Each *IportBuffer* will associate to a Boolean variable *NewData* which represents the input buffer has got the latest data. It is used to guarantee the deadline of the sender will be before the dispatch of the recipient when the connection type between them is *Delayed*. *CurrentTime* represents the global current time.

$$
\begin{aligned}
S_{\mathcal{G}} = & \bigcup_{\substack{th\in \text{Threads}(m),\\ ip\in \text{Iports}(th)}} \{\text{IportBuffer}(th, ip) : \text{VALUE}\} \\
& \cup \bigcup_{\substack{th\in \text{Threads}(m),\\ ip\in \text{Iports}(th)}} \{\text{NewData}(th, ip) : \text{BOOL}\} \\
& \cup \bigcup_{\substack{th\in \text{Threads}(m),\\ op\in \text{Oports}(th)}} \{\text{Oport}(th, op) : \text{VALUE}\} \\
& \cup \{\text{CurrentTime} : \text{DURATION}\}
\end{aligned}
$$

- $\boldsymbol{init}(S_{\mathcal{G}}) =$

$$
\begin{aligned}
& \{\text{IportBuffer}(th, ip) \mapsto 0 \mid th \in \text{Threads}(m) \ \wedge \ ip \in \text{Iports}(th)\} \\
\cup & \{\text{NewData}(th, ip) \mapsto \mathit{true} \mid th \in \text{Threads}(m) \ \wedge \ ip \in \text{Iports}(th)\} \\
\cup & \{\text{Oport}(th, op) \mapsto 0 \mid \ th \in \text{Threads}(m) \ \wedge \ op \in \text{Oports}(th)\} \\
\cup & \{\text{CurrentTime} \mapsto 0\}
\end{aligned}
$$

  Here, messages exchanged by threads are supposed to be of type Integer.
- The merging of the two global states $g_1, g_2 \in S_{\mathcal{G}}$ is the partial function defined as follows: $\text{mrg}(g_1, g_2) =$

$$
\left\{ v \mapsto \begin{array}{ll} g_1(v) & \text{if } v \in \text{dom}(g_1) \setminus \text{dom}(g_2) \\ g_2(v) & \text{if } v \in \text{dom}(g_2) \setminus \text{dom}(g_1) \\ g_1(v) & \text{if } v \in \text{dom}(g_1) \cap \text{dom}(g_2) \wedge g_1(v) = g_2(v) \end{array} \middle| \ v \in \text{dom}(S_{\mathcal{G}}) \right\}
$$

  Note that, merging is undefined if the global state disagrees on the value of a shared variables.

Definition 9 *($TTS_{periodic\_thread}$).* The behavior of a periodic thread th is a timed transition system $\text{TTS}_{periodic\_thread}(th) = \langle L_{\mathcal{L}} \cup \mathbb{R}^+, S_{\mathcal{L}}, \boldsymbol{init}, \boldsymbol{t\_next} \rangle$ defined over $L_{\mathcal{G}}$ and $S_{\mathcal{G}}$ where:

- $L_{\mathcal{L}} = \{\text{dispatch}(th), \text{output}(th)\}$.
- $S_{\mathcal{L}} = \{\text{State}(th) : \{ \mathit{waiting\_dispatch}, \mathit{waiting\_execution}, \mathit{execution}, \mathit{computed}, \mathit{completed}\},$
  $\text{Iport}(th) : \text{Iports}(th) \rightarrow \text{VALUE},$
  $\text{StartofExeTime}(th) : \text{DURATION},$
  $\text{DispatchTime}(th) : \text{DURATION}\}.$

As mentioned above, for an immediate connection, the execution of the recipient is suspended until the sender completes its execution. So we save the start time of execution of the thread. Moreover, we save the dispatch time of the thread.

- $\textbf{\textit{init}}(S_{\mathcal{L}}) = \{\text{State}(th) \mapsto \textit{waiting\_dispatch},$
  $\text{Iport}(th) \mapsto \{ip \mapsto 0 \mid ip \in \text{Iports}(th)\},$
  $\text{StartofExeTime}(th) \mapsto 0,$
  $\text{DispatchTime}(th) \mapsto 0\}.$
- ***t_next*** is defined by the following rules, including discrete transitions and delay transitions. s and s' are values of the type S (defined as $\{(g, l) \mid g \in S_{\mathcal{G}} \wedge l \in S_{\mathcal{L}}^{g}\}$). The schema $s' = s$ *with* $\{\ldots\}$ specifies the new value of the updated fields. We also use the overloading operator $\lhd\!\!+$.

DISPATCH. This rule represents the dispatch of a thread. On the *dispatch* event, the input ports are read. For this purpose, we must wait for the data transfer made by the sender threads through delayed connections when their deadline occurs at the same logical time as the dispatch of the recipient thread. This condition is defined by the *DelayedSyncCond* predicate.

$$\frac{\begin{array}{l} s(\textit{State}(th)) = \textit{waiting\_dispatch}, \\ \textit{DelayedSyncCond}(th, s), \\ s' = s \text{ with } \{ \\ \quad \textit{State}(th) = \textit{waiting\_execution}, \\ \quad \textit{DispatchTime}(th) = s(\textit{CurrentTime}), \\ \quad \textit{Iport}(th) = \{ip \mapsto s(\textit{IportBuffer}(th, ip)) \mid ip \in \textit{Iports}(th)\}, \\ \quad \|_{ip \in \textit{Iports}(th)}\ \textit{NewData}(th, ip) = \textit{false} \\ \} \end{array}}{s \xrightarrow{\textit{dispatch}(th)} s'}\ \textit{DISPATCH}$$

where $\text{DelayedSyncCond}(th, s) \equiv$

$$\begin{array}{l} \forall cn \in \text{Connections}(m), \\ \quad s(\textit{DispatchTime}(\textit{SrcThread}(cn))) + \textit{Period}(\textit{SrcThread}(cn)) \\ \qquad = s(\textit{CurrentTime}) \\ \wedge\ \textit{DstThread}(cn) = th \\ \wedge\ \textit{ConnectionType}(cn) = \textit{Delayed} \\ \Rightarrow s(\textit{NewData}(th, \textit{DstPort}(cn))) = \textit{true} \end{array}$$

WAITING_EXE. When the thread is dispatched, its execution is managed by a scheduler. For example, in presence of immediate connection, the scheduler must ensure that the sender completes before the start of execution of the recipient. Moreover, ports of which incoming connection is immediate must be rewritten. The rule *WAITING_EXE[1]* means that the thread gets CPU, while the rule *WAITING_EXE[2]* represents the thread waits to be scheduled. The delay is controlled by a TTS associated with the scheduler.

$$\frac{\begin{array}{l} s(\textit{State}(th)) = \textit{waiting\_execution}, \\ s' = s \text{ with } \{ \\ \quad \textit{State}(th) = \textit{execution}, \\ \quad \textit{Iport}(th) = s(\textit{Iport}(th)) \lhd\!\!+ \\ \qquad \{ip \mapsto s(\textit{IportBuffer}(th, ip)) \mid \text{isImmediate}(th, ip)\}, \\ \quad \textit{StartofExeTime}(th) = s(\textit{CurrentTime}) \\ \} \end{array}}{s \xrightarrow{\textit{execute}(th)} s'}\ \textit{WAITING\_EXE}[1]$$

where $\text{isImmediate}(th, ip)$ checks if the input port ip of thread th is the destination of an immediate connection. This predicate is defined as follows:

$$\begin{array}{l} \exists cn \in \textit{Connections}(m),\ \textit{DstPort}(cn) = ip \\ \qquad\qquad\qquad\qquad \wedge\ \textit{ConnectionType}(cn) = \textit{Immediate} \\ \qquad\qquad\qquad\qquad \wedge\ ip \in \textit{Iports}(th) \end{array}$$

$$\frac{\begin{array}{l} s(\textit{State}(th)) = \textit{waiting\_execution}, \\ s' = s \text{ with } \{\textit{CurrentTime} = s(\textit{CurrentTime}) + \delta\} \end{array}}{s \xrightarrow{\delta} s'}\ \textit{WAITING\_EXE}[2]$$

EXECUTION. The execution is abstracted by the function *ComputeOutput* which consumes CPU time during the interval [BCET, WCET] of the thread, and it can be refined by the AADL behavior annex. The rule *EXECUTION[1]* represents that the

thread is in the progress of its execution. The rule *EXECUTION[2]* is used to write the computation results into the output ports of the thread. The rule *EXECUTION[3]* means that the thread enters the *completed* state after all the outputs are written.

$$\frac{\begin{array}{l} s(State(th)) = execution, \\ s(CurrentTime) - s(StartofExeTime(th)) + \delta \leq th.WCET, \\ s' = s \text{ with } \{CurrentTime = s(CurrentTime) + \delta\} \end{array}}{s \xrightarrow{\delta} s'} EXECUTION[1]$$

$$\frac{\begin{array}{l} s(State(th)) = execution, \\ s(CurrentTime) - s(StartofExeTime(th)) \\ \quad \in th.BCET..th.WCET, \\ s' = s \text{ with } \{ \\ \quad State(th) = computed, \\ \quad \|_{op \in Oports(th)}\ Oport(th, op) = ComputeOutput \\ \qquad\qquad (\{ip \mapsto s(Iport(th)) \mid ip \in Iports(th)\}, op) \\ \} \end{array}}{s \xrightarrow{output(th)} s'} EXECUTION[2]$$

$$\frac{\begin{array}{l} s(State(th)) = computed, \\ s' = s \text{ with } \{State(th) = completed\} \end{array}}{s \xrightarrow{complete(th)} s'} EXECUTION[3]$$

WAITING_DEADLINE. This rule is used to wait for the deadline (i.e., the period here for the sake of simplicity) of the thread.

$$\frac{\begin{array}{l} s(State(th)) = completed, \\ s(CurrentTime) + \delta \\ \quad \leq s(DispatchTime(th)) + Period(th), \\ s' = s \text{ with } \{CurrentTime = s(CurrentTime) + \delta\} \end{array}}{s \xrightarrow{\delta} s'} WAITING\_DEADLINE$$

DEADLINE. This rule represents that the thread enters the next dispatch.

$$\frac{\begin{array}{l} s(State(th)) = completed, \\ s(CurrentTime) = s(DispatchTime(th)) + Period(th), \\ s' = s \text{ with } \{State(th) = waiting\_dispatch\} \end{array}}{s \xrightarrow{deadline(th)} s'} DEADLINE$$

Definition 10 *($TTS_{connection}$).* The behavior of a data-port connection cn is a timed transition system $\text{TTS}_{connection}(cn) = \langle L_{\mathcal{L}} \cup \mathbb{R}^+, S_{\mathcal{L}}, \boldsymbol{init}, \boldsymbol{t\_next} \rangle$ defined over $L_{\mathcal{G}}$ and $S_{\mathcal{G}}$ where:

- $L_{\mathcal{L}} = \emptyset$.
- $S_{\mathcal{L}} = \{\text{CnState}(cn) : \{ready\}\}$.
- $\boldsymbol{init}(S_{\mathcal{L}})$ is considered as $\{\text{CnState}(cn) \mapsto ready\}$.
- ***t_next*** is defined by the following rules.

WRITE_IMM_DATA. This rule synchronizes with the rule *EXECUTION[3]* of $\text{TTS}_{periodic\_thread}(th)$. It is used to copy the data of the output ports of the sender to the *IportBuffer* of the recipient at the completion time of the sender. This copy is only done when the two threads are dispatched at the same time.

$$\frac{\begin{array}{l} \text{ConnectionType}(cn) = \text{Immediate}, \\ s(DispatchTime(SrcThread(cn))) = \\ \quad s(DispatchTime(DstThread(cn))), \\ s' = s \text{ with } \{ \\ \quad IportBuffer(DstThread(cn), DstPort(cn))) = \\ \quad\ s(Oport(SrcThread(cn), SrcPort(cn))) \\ \} \end{array}}{s \xrightarrow{complete(th)} s'} WRITE\_IMM\_DATA$$

WRITE_DEL_DATA. This rule synchronizes with the rule *DEADLINE* of $\text{TTS}_{periodic\_thread}(th)$. It is used to copy the data of the output ports of the sender to the *IportBuffer* of the recipient at the deadline of the sender. We also set the *NewData* variable to true to indicate the data transfer has been done.

$$\frac{\begin{array}{l}\text{ConnectionType}(cn) = \text{Delayed},\\ s' = s \text{ with } \{\\ \quad IportBuffer(DstThread(cn), DstPort(cn))) =\\ \quad\quad s(Oport(SrcThread(cn), SrcPort(cn))),\\ \quad NewData(DstThread(cn), DstPort(cn))) = true\\ \}\end{array}}{s \xrightarrow{deadline(th)} s'}\ WRITE\_DEL\_DATA$$

Timed transition systems appear to be well suited to describe the formal semantics of AADL. The subset considered in this section can be extended in the future.

6. Bisimulation discussion

In order to enable the proof of semantics preservation of the model transformation (see Fig. 8), we formalize the subset of AADL using TTS and consider it as a reference semantics. As the behavior is explicitly expressed by atomic transitions, the TTS-based semantics is easy to understand. However, TTS*s* are mainly a mathematical model which cannot be used to automatically verify properties of a given AADL model. FIACRE-based semantics (for verification purpose) is more complex than the TTS-based one, because we need to know the rather complex FIACRE semantics to understand the translated FIACRE sentences. The TTS-based semantics is assumed to be correct, and it gives a reference expression of AADL semantics which can be compared with the FIACRE-based semantics. This will ultimately allow the formal verification of the toolchain by comparing through bisimulation two TTS-based semantics of AADL.

- *Mechanization of the transformation.* As shown in previous sections, we have mechanized the operational semantics of the reduced subset of AADL in TTS which is named AADL_tts and the operational semantics of FIACRE. Moreover, we will mechanize the transformation from the reduced subset of AADL to FIACRE. Thus, we get the second TTS which is named FIACRE_TTS. It is the TTS corresponding to the semantics of the FIACRE model obtained by translating the considered AADL model.
- *Compositional proof methodology.* The composition of TTS*s* is used to define the semantics of an AADL model as the composition of the semantics of its constituents. This way, the equivalence proof can be obtained in a compositional way from the correctness of the translation of elementary AADL model elements such as threads, connections, etc. For example, given an AADL model which includes several periodic threads with delayed connections. We would like to prove the bisimulation between the operational semantics of a thread and its translation into FIACRE, and then the full semantics will be preserved by using composition in both sides. This method is called as compositional proof and it will reduce the complexity of the proof.
- *The extraction of a verified tool chain.* As shown in Part 1, the model transformation tool has been implemented. However, it is a good way to provide a basis for the COQ mechanization. Finally, we envision the extraction of a new tool from the mechanization in the future, that is the verified toolchain.

7. Related works

In the following, we review some works which aim at verifying AADL models, its correctness. When described using an AADL model, such a system specification is often transformed to another formal model for verification and analysis. Examples of such transformations are numerous: translations to Behavior Interaction Priority (BIP) [11], to real-time process algebra ACSR [41], to Real-Time Maude [33], to Lustre [28], to Polychrony [32], etc.

The authors of [11] translate most of the AADL concepts into the BIP language, except for the mode change. BIP is a framework for modeling heterogeneous real-time components using three layers: the lower layer describes behaviors, the intermediate layer describes interactions, and the upper layer describes scheduling policies. It provides two categories of verification properties: deadlock detection by using the tool Aldebaran, and some simple timing properties using observers.

The translation proposed by [41] mainly focuses on the schedulability analysis of AADL models. A smaller subset (modes and the behavior annex are excluded) is translated into the real-time process algebra ACSR. The toolset uses the ACSR-based tool VERSA to explore the state space of the model, looking for violations of timing requirements. ACSR can explicitly model resource usages.

In the work of [33], a subset of AADL is translated into Real-Time Maude, and it provides an AADL simulator and LTL model checking tool called AADL2Maude. Real-Time Maude is based on a formal real-time rewriting logic.

The translation proposed by [28] covers a subset of AADL except for the behavior annex, and is instead given as a program in the synchronous language Lustre. The translation into Polychrony [32] mainly focuses on the multi-partitions structure of an embedded architecture and aims at simulating and verifying a GALS (globally asynchronous locally synchronous) model from the given AADL specifications.

Monteverde et al. [31] propose Visual Timed Scenarios (VTS) as a graphical language for the specification of AADL behavioral properties. Then a translation from VTS to Timed Petri Nets is presented. Model-checking of properties expressed in VTS is enabled using TPN-based tools.

Rugina et al. [36] propose a dependability analysis tool (called ADAPT) on the AADL models. Its input is an AADL model annotated with dependability-related information (through the AADL error model annex), and its output is a dependability evaluation model in the form of a Generalized Stochastic Petri Net (GSPN). The GSPN model can be processed by existing dependability evaluation tools, to compute quantitative measures such as reliability, availability, etc.

The authors of [8,7] present a graphical toolset, called the COMPASS platform, for verifying AADL models by capturing functional, probabilistic and hybrid aspects. Analyses are implemented on top of mature model checking tools and range from requirements validation to functional verification, safety assessment via automatic derivation of FMEA (Failure Mode and Effects Analysis) tables and dynamic fault trees, to performability evaluation, and diagnosability analysis.

The problem of formally analyzing AADL models is an absolutely crucial problem that has been addressed by a number of research groups recently. Most of those efforts have targeted subsets of the large AADL language for functional analysis, schedulability analysis, or dependability analysis, etc. In this work, we consider a behavioral subset for functional analysis and schedulability analysis.

8. Conclusion

In this paper, we have given an overview of the foundational aspects of the verification support for AADL. This work has been actually experimented in the verification toolchain for AADL that has been made available in the TOPCASED environment. Two major versions have been elaborated. In the first one [5], we were mainly concerned by the translation process in the emerging model driven engineering process based on the EMF technology [40]. This work was based on the metamodel for AADL provided by the OSATE [10] tool, which is integrated in TOPCASED and the metamodel for FIACRE also available in the TOPCASED environment. In the second version, presented here, we have been concerned by semantics aspects. Essentially, we have addressed the so-called synchronous subset. We have elaborated a translation from that subset to the FIACRE language. In order to make easier the validation of such a translation, an enhancement of the basic FIACRE language, called FIACRE* have been introduced together with a real-time library. We are convinced that these two extensions are essential for the proof process. At last, we have given the main aspects of the semantics domains we have used.

With respect to our experimentation, we draw the following conclusions:

- Considering the AADL to FIACRE* transformation, we have used a dedicated model to text language which has shown some limits for dealing with separators and in general the concrete syntax of the target language. On the other hand, the model to model approach, based on the abstract syntax, is hard to use because it requires to understand the internal representation of syntactic elements. A mixed approach [14] would be more appropriate, where instances of the abstract syntax would be built using concrete syntax fragments.
- Considering model checking of high level modeling languages, while it is attractive to reuse existing model checkers to benefit from their reliability and performance, the diagnostic they deliver is hard to understand and may refer to computation steps that are not relevant for the end user. Thus, a co-design of the language transformation together with a reverse trace transformation [45] has to be integrated (and of course, its correctness proof too!).
- Considering semantical aspects, we have presented semantics domains related to real-time. However, the mechanization of such domains is not straightforward and raises many questions. As it is shown by our work, as well by related works about mechanization of transitions systems-based domains, standard definitions have yet to be established. For instance, we have chosen to distinguish local labels from global ones in TTS, which is very useful for component-based languages where local labels are observable.
- Concerning AADL, we are aware that one should remain modest with respect to the coverage of the language. Among the key aspects that we have not taken into account, let us mention he notion of modes, which is found in a number of architecture description languages like GIOTTO [20] and AADL. Our first investigations reveal that such a feature would probably entail to reconsider the FIACRE pivot language in order to introduce well suited mechanisms. Another aspect that is also worth to consider is high level formal specification of AADL behaviors [2] starting from AADL requirements [9]. With respect to that, the contract concept [34] looks promising A related issue worth to mention is a better support for error analysis of such specifications. Last but not least, it is clear that the “raison d’être” of all this semantics work is a more reliable code generation towards certified code [26]. Here also, foundational work has already been undertaken [25].

Acknowledgements

This work has been mainly sponsored by the projects TOPCASED and QUARTEFT. The authors thank the members of these projects that have also participated in this work (Patrick Farail and Pierre Gaufillet from Airbus, Pierre Dissaux from Ellidis, Bernard Berthomieu, Silvano Dal Zilio and François Vernadat from LAAS CNRS).

Moreover, the authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper.

References

[1] Thomas Abdoul, Joël Champeau, Philippe Dhaussy, Pierre Yves Pillain, Jean-Charles Roger, AADL execution semantics transformation for formal verification, in: ICECCS, IEEE Computer Society, 2008, pp. 263–268.

[2] Nouha Abid, Silvano Dal Zilio, Didier Le Botlan, Real-time specification patterns and tools, in: 17th International Workshop on Formal Methods for Industrial Critical Systems, FMICS 2012, in: Lecture Notes in Computer Science, vol. 7437, Springer-Verlag, September 2012, pp. 1–15.

[3] AESE, Aerospace valley, http://www.aerospace-valley.com/.

[4] André Arnold, Finite Transition Systems – Semantics of Communicating Systems, Prentice Hall International Series in Computer Science, Prentice Hall, 1994.

[5] Bernard Berthomieu, Jean-Paul Bodeveix, Christelle Chaudet, Silvano Dal-Zilio, Mamoun Filali, François Vernadat, Formal verification of AADL specifications in the topcased environment, in: Fabrice Kordon, Yvon Kermarrec (Eds.), Ada-Europe, in: Lecture Notes in Computer Science, vol. 5570, Springer, 2009, pp. 207–221.

[6] Yves Bertot, Pierre Castéran, Interactive Theorem Proving and Program Development (CoqArt: The Calculus of Inductive Constructions). Texts in Theoretical Computer Science, Springer, 2004.

[7] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, Marco Roveri, Ralf Wimmer, A model checker for AADL, in: Tayssir Touili, Byron Cook, Paul Jackson (Eds.), CAV, in: Lecture Notes in Computer Science, vol. 6174, Springer, 2010, pp. 562–565.

[8] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, Marco Roveri, Safety, dependability and performance analysis of extended AADL models, Comput. J. 54 (5) (2011) 754–775.

[9] Dominique Blouin, Eric Senn, Skander Turki, Defining an annex language to the architecture analysis and design language for requirements engineering activities support, in: MoDRE, IEEE, 2011, pp. 11–20.

[10] CMU. An Extensible Open Source AADL Tool Environment (OSATE), 2006.

[11] Mohamed Yassin Chkouri, Anne Robert, Marius Bozga, Joseph Sifakis, Translating AADL into BIP – application to the verification of real-time systems, in: Michel R.V. Chaudron (Ed.), MoDELS Workshops, in: Lecture Notes in Computer Science, vol. 5421, Springer, 2008, pp. 5–19.

[12] Mamoun Filali-Amine, Julia L. Lawall, Development of a synchronous subset of AADL, in: Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, Steve Reeves (Eds.), ASM, in: Lecture Notes in Computer Science, vol. 5977, Springer, 2010, pp. 245–258.

[13] Ricardo Bedin França, Jean-Paul Bodeveix, Mamoun Filali, Jean-François Rolland, David Chemouil, Dave Thomas, The AADL behaviour annex – experiments and roadmap, in: ICECCS, IEEE Computer Society, 2007, pp. 377–382.

[14] Elie Fares, Jean-Paul Bodeveix, Mamoun Filali, Design of a BPEL verification tool, in: Marco Carbone, Jean-Marc Petit (Eds.), WS–FM'2011, in: Lecture Notes in Computer Science, vol. 7176, Springer, 2011, pp. 95–110.

[15] Peter H. Feiler, David P. Gluch, Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language, Addison-Wesley, 2013.

[16] Patrick Farail, Pierre Gaufillet, Florent Peres, Jean-Paul Bodeveix, Mamoun Filali, Bernard Berthomieu, Saad Rodrigo, François Vernadat, Hubert Garavel, Frédéric Lang, FIACRE: an intermediate language for model verification in the TOPCASED environment, in: European Congress on Embedded Real-Time Software, ERTS, Toulouse, 29/01/2008–01/02/2008, Société de l'Electricité, de l'Electronique et des Technologies de l'Information et de la Communication (SEE), janvier 2008, http://www.see.asso.fr.

[17] The Eclipse Foundation. Acceleo – transforming models into code, http://www.eclipse.org/acceleo/.

[18] Manuel Garnacho, Jean-Paul Bodeveix, Mamoun Filali-Amine, A mechanized semantic framework for real-time systems, in: FORMATS, 2013, pp. 106–120.

[19] Gregor Gößler, Joseph Sifakis Priority systems, in: Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, Willem P. de Roever (Eds.), FMCO, in: Lecture Notes in Computer Science, vol. 3188, Springer, 2003, pp. 314–329.

[20] T.A. Henzinger, C.M. Kirsch, M.A.A. Sanvido, W. Pree, From control models to real-time code using Giotto, IEEE Control Syst. 23 (1) (2003) 50–64.

[21] Thomas A. Henzinger, Zohar Manna, Amir Pnueli, Temporal proof methodologies for real-time systems, in: POPL, 1991, pp. 353–366.

[22] Thomas A. Henzinger, Zohar Manna, Amir Pnueli, Timed transition systems, in: REX Workshop, 1991, pp. 226–251.

[23] Thomas A. Henzinger, Zohar Manna, Amir Pnueli, Temporal proof methodologies for timed transition systems, Inf. Comput. 112 (1994) 273–337.

[24] C.A.R. Hoare, Communicating Sequential Processes, 1985.

[25] Jérôme Hugues, Bechir Zalila, Laurent Pautet, Fabrice Kordon, From the prototype to the final embedded system using the ocarina AADL tool suite, ACM Trans. Embed. Comput. Syst. 7 (4) (2008).

[26] Nassima Izerrouken, Olivier Ssi Yan Kai, Marc Pantel, Xavier Thirioux, Use of formal methods for building qualified code generator for safer automotive systems, in: Jean–Charles Fabre, Olivier Guetta, Mario Trapp (Eds.), EDCC–CARS, in: ACM International Conference Proceeding Series, vol. 10, ACM, 2010, pp. 53–56.

[27] The ISABELLE System, http://isabelle.in.tum.de/.

[28] Erwan Jahier, Nicolas Halbwachs, Pascal Raymond, Xavier Nicollin, David Lesens, Virtual execution of AADL models via a translation into synchronous programs, in: Christoph M. Kirsch, Reinhard Wilhelm (Eds.), EMSOFT, ACM, 2007, pp. 134–143.

[29] LORIA, A software environment for defining transformations, http://tom.loria.fr/.

[30] Robin Milner, Communication and Concurrency, PHI Series in Computer Science, Prentice Hall, 1989.

[31] Daniel Monteverde, Alfredo Olivero, Sergio Yovine, Víctor Braberman, VTS-based specification and verification of behavioral properties of AADL models, in: Model Based Architecting and Construction of Embedded Systems, ArtistDesign European Network of Excellence on Embedded Systems Design, Toulouse, France, dic 2008.

[32] Yue Ma, Jean-Pierre Talpin, Sandeep K. Shukla, Thierry Gautier, Distributed simulation of AADL specifications in a polychronous model of computation, in: Tianzhou Chen, Dimitrios N. Serpanos, Walid Taha (Eds.), ICESS, IEEE, 2009, pp. 607–614.

[33] Peter Csaba Ölveczky, Artur Boronat, José Meseguer, Formal semantics and analysis of behavioral AADL models in real-time Maude, in: John Hatcliff, Elena Zucca (Eds.), FMOODS/FORTE, in: Lecture Notes in Computer Science, vol. 6117, Springer, 2010, pp. 47–62.

[34] J.-B. Raclet, E. Badouel, A. Benveniste, B. Caillaud, A. Legay, R. Passerone, A modal interface theory for component-based design, Fundam. Inform. 108 (1–2) (2011) 119–149.

[35] Jean-François Rolland, Jean-Paul Bodeveix, Mamoun Filali, David Chemouil, Dave Thomas, Modes in asynchronous systems, in: ICECCS, IEEE Computer Society, 2008, pp. 282–287.

[36] Ana-Elena Rugina, Karama Kanoun, Mohamed Kaâniche, The ADAPT tool: from AADL architectural models to stochastic Petri nets through model transformation, in: EDCC, IEEE Computer Society, 2008, pp. 85–90.

[37] John Rushby, Mechanized formal methods: progress and prospects, in: Proc. of FSTTCS, Hyderabad, India, in: LNCS, vol. 1180, 1996, pp. 43–51.

[38] SAE, AS5506A: Architecture Analysis and Design Language (AADL) Version 2.0, 2009.

[39] SAE, AS5506 Annex: Behavior_Specification V2.0, 2011.

[40] David Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks, EMF: Eclipse Modeling Framework 2.0, 2nd edition, Addison-Wesley Professional, 2009.

[41] Oleg Sokolsky, Insup Lee, Duncan Clarke, Schedulability analysis of AADL models, in: IPDPS, IEEE, 2006.
[42] Régis Spadotti, Gabriel Santos, Jean-Paul Bodeveix, Mamoun Filali, APOTA case study, http://www.irit.fr/~Regis.Spadotti/AADL2FIACRE/APOTA.zip.
[43] TOPCASED, The open-source toolkit for critical systems, http://www.topcased.org/.
[44] Silvano Dal Zilio, Bernard Berthomieu, Jean-Paul Bodeveix, Mamoun Filali, Didier Le Botlan, Guillaume Verdier, François Vernadat, Real-time model checking support for AADL, Technical report 15036, LAAS, March 2015.
[45] Faiez Zalila, Xavier Crégut, Marc Pantel, A transformation-driven approach to automate feedback verification results, in: Alfredo Cuzzocrea, Sofian Maabout (Eds.), MEDI, in: Lecture Notes in Computer Science, vol. 8216, Springer, 2013, pp. 266–277.