Formal Verification of AADL Specifications in the Topcased Environment*

B. Berthomieu[1,3], J.-P. Bodeveix[2,3], C. Chaudet[2,3],
S. Dal Zilio[1,3], M. Filali[2,3], and F. Vernadat[1,3]

[1] CNRS ; LAAS ; 7 avenue colonel Roche, F-31077 Toulouse, France
[2] CNRS ; IRIT ; Université de Toulouse, 118 route de Narbonne, F-31062 Toulouse, France
[3] Université de Toulouse ; UPS, INSA, INP, ISAE, UT1, UTM ; F-31062 Toulouse, France

Abstract. We describe a formal verification toolchain for AADL, the SAE Architecture Analysis and Design Language, enriched with its behavioral annex. Our approach is based on tools that are integrated in the Topcased environment. We give a high-level view of the tools involved and illustrate the successive transformations that take place during the verification process.

1 Introduction

Aeronautics and space are sectors that produce and rely on complex, critical, hardware and software systems. These systems typically have a long life cycle of development and maintenance, spanning several decades. In this process, design is the main phase of a project since code, tests and documentations are generated from the models. Moreover, the tools supporting the design process should be reliable and readily available, so as to avoid obsolescence or technology lock-in.

The Architecture and Analysis Design Language (AADL) [1] is a standard, promoted by the Society of Automotive Engineers (SAE), for the specification and analysis of complex real-time embedded systems. AADL is a textual and graphical language, designed for model-based engineering, which can be used to describe both the software and hardware components of a system.

In order to support model-based development, companies from the French Aeronautics, Space and Embedded Systems competitivity pole (AESE) have joined their efforts to develop a common set of methods and tools. The goal is to deliver an industrial strength system/software development platform for embedded systems. The Topcased [18] initiative is part of this effort and AADL is among the first languages supported in this project. Topcased is also the name of a toolkit based on the Eclipse platform and concepts that provides an open source, model oriented set of tooling and standard implementations.

Our main objective in this paper is to present a verification toolchain for AADL specifications that is fully integrated in the Topcased environment. We give a high-level view of the tools involved and illustrate the successive transformations required by

* This work was partly supported by the French AESE project Topcased, The ANR project OpenEmbeDD, and by region Midi-Pyrénées.

our verification process. These tools include a semantic editor of AADL and a model-checking platform based on Time Petri-Net, Tina [11]. The generation of Tina models from an AADL description relies on the Fiacre intermediate language. First, the AADL description is translated in the Fiacre format, which offers a formal intermediate model to represent both the behavioral and timing aspects of the system. Then, we compile the Fiacre model into an abstract Timed Transition Systems that can be directly analyzed by Tina. Therefore, Fiacre is designed both as the target language of model transformation engines from various models such as SDL, UML, AADL, ... and as a front end to targeted verification toolboxes, namely CADP and Tina in the first step.

After a brief review of AADL, we present the Fiacre language and describe the fundamentals of our translation from AADL to Fiacre. The transformation is illustrated on a small example. Before concluding, we introduce the Tina verification platform and give examples of properties that can be verified on the models.

2 AADL

The SAE Architecture and Analysis Design Language is a standard used to describe both the software and hardware components of a system. Specifications based on an architecture description language have the benefit to provide support for an early analysis of the properties of a system. These properties are supposed to be obtained before the coding or the actual deployment of the system. Model-based system engineering (MBE) lets you focus on the analysis of system architecture — and detect problems with availability, security, and timeliness early on. AADL descriptions pertain to the functional interfaces of components, such as data inputs and outputs, as well as to non-functional aspects, such as timing properties. The language can describe how components are combined, such as how data inputs and outputs are connected or how software components are allocated to hardware components.

Release 1.0 of the AADL standard (SAE AS5506) has been issued in November 2004. Since then, many extensions have been proposed. Some of them, like the Error Model Annex, have been adopted by the standardization committee. Tools have also been developed, like the initial OSATE environment, which has been merged with Topcased.

AADL includes all the standard concepts of an Architecture Description Language: components, connectors (used to describe the interface of components), and connections (used to link components). The set of AADL components can be divided in three partitions, the software components (process, thread, thread group, subprogram, and data), the hardware components (processor, bus, memory, device), and system components. Components can communicate through ports, synchronous calls, and shared data. A process represents a virtual address space, or a partition; this address space includes the program defined by its sub-components.

A process must contain at least one thread or thread group. A thread group is a logical organization of threads in a process. A thread represents a sequential flow of execution; it is the only AADL component that can be scheduled. A subprogram represents a piece of code that can be called by a thread or another program. A data models a static variable used in the code; threads and processes can share data.

An AADL model can incorporate non-architectural elements: embedded or real-time characteristics of the components (execution time, memory usage, etc.), behavioral descriptions, etc. Hence it is possible to use AADL as a backbone to describe all the aspects of a system. To this end, AADL defines the notion of properties that can be attached to most elements (components, connections, features, etc.). Properties are attributes that specify constraints or characteristics that apply to the elements of the architecture: clock frequency of a processor, execution time of a thread, bandwidth of a bus, etc. Some standard properties are defined but the definition can be extended with user-defined properties.

Finally, AADL supports an annex mechanism to extend the description capabilities of the language by introducing a dedicated sub-language. A behavior annex [4] is currently being defined by the SAE committee.

2.1 The Execution Model

Threads are the only components that have an execution semantics. AADL supports the classical types of dispatch protocols: a thread can be declared as periodic, aperiodic, sporadic or background. All the standard properties (WCET, deadline, ...) used to describe a real-time system exist in AADL. Threads have two predeclared event ports : dispatch and complete. The dispatch port is used for aperiodic or sporadic threads. If this port is connected all other ports of the thread do not trigger the dispatch. It is a very common behavior for an aperiodic or a sporadic thread to send an event on completion. In AADL, we do not specify when an event is sent. The complete event port is used to send an event at the end of the execution.

All threads have the same life cycle. This cycle can be represented as an automaton (see Fig. 1). All threads start in the awaiting_dispatch state. The dispatch condition depends on the thread type. If the thread is periodic it will be dispatched at every period. At this time, delivery occurs for all its input ports. An aperiodic or a sporadic thread that does not have its dispatch ports connected is dispatched each time it receives an event. Delivery occurs only for the port that triggers the dispatch and the data ports. If its dispatch port is connected, it is dispatched each time it receives an event on this port, and delivery occurs for all its others ports. The thread in the active state that has the maximum priority starts or continues its execution. The priority of the thread is determined by the chosen scheduling policy (RMA, EDF, LLF). This policy is specified by a property of the model. When a thread is dispatched it can have a higher priority than the executing thread. In this case, the executing thread is preempted and goes back to the active state. When a thread ends its execution it goes to the awaiting_dispatch state until the next dispatch. At this time, all the output data ports of the thread are read and their content sent to the destination ports.

The scheduling behavior we just described is slightly different if shared resources are used. When an executing thread tries to access a locked shared resource, it goes to a special state, labeled awaiting_resource in our automata, where it blocks until the resource (its lock) is released. The process by which a variable is locked depends on the kind of implementation used for the AADL specification. In the most general case,

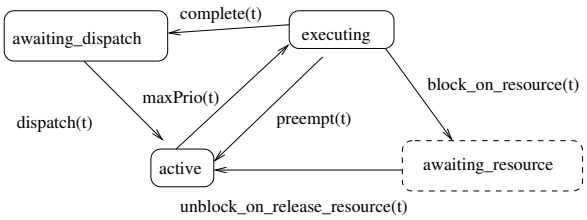

Fig. 1. *Thread automaton with (and without) shared resources.*

the basic implementation is to lock a shared resource during the whole execution of the thread that accesses it.

2.2 A Toy Example

In this section, we illustrate AADL by encoding a simple token ring example (Listing 1.1). It is a network of processes (AADL threads), logically organized in a ring topology, which synchronize their communication by means of a *token* that circulates among them, controlling access to the communication channels. The goal is to implement mutual exclusion among all threads.

We suppose that threads communicate only through ports: each thread is connected to a successor (resp. predecessor) thread by its port named `succ` (resp. `pred`). A thread enters mutual exclusion once it owns the token. The proposed AADL model consists in a thread `Start` that chooses non-deterministically the `Node` that initially owns the token (actions `start0!`, ..., `start2!`). Each node communicates with its predecessor and successor nodes over its ports named `prev` and `succ`. Each thread is supposed to be sporadic with a minimal period of 10ms. The behavior of a node is described through the behavioral annex. Initially, threads are in the `idle` state. When it receives the token (action `prev?`), it forwards it (action `succ!`) then it can, non-deterministically, either remain idle or become waiting. When a waiting thread receives the token, it enters mutual exclusion (state `cs`). After a computation, which can elapse between 5 and 10 ms, it leaves mutual exclusion, transfers the token to its neighbour and becomes idle again.

3 AADL to Fiacre Translation

The modeling language of model checkers must be kept simple enough for the tool to be efficient. It is therefore preferable to reuse existing model checkers (e.g. Tina [11] and CADP [12]) to check properties of high-level languages (UML, SysML, AADL), which leads to the introduction of a transformation phase between the two levels. In order to simplify the connection of model checkers to those languages, we have introduced the pivot language Fiacre [13] so that the expression of the run-time semantics of high-level languages can be factorized and expressed using the pivot. It must be powerful enough

```
thread Start
  features start0: out event port; start1: out event port;
           start2: out event port;
  properties Dispatch_Protocol => Background;
end Start;

thread implementation Start.i
  annex behavior_specification {**
    states  s0: initial state; s1: complete state;
    transitions  s0 -[]-> s1 { start0!; };
    transitions  s0 -[]-> s1 { start1!; };
    transitions  s0 -[]-> s1 { start2!; };
  **};
end Start.i;

thread Node
  features prev: in event port; succ: out event port;
           start: in event port;
  properties Dispatch_Protocol => Sporadic; Period => 10ms;
end Node;

thread implementation Node.i
  annex behavior_specification {**
    states idle: initial complete state;
           wait: complete state;
           cs: state;
    transitions
      idle -[start?]-> idle { succ!; };
      idle -[prev?]-> idle { computation(3ms); succ!; };
      idle -[prev?]-> wait { computation(3ms); succ!; };
      wait -[prev?]-> cs;
      cs -[]-> idle { computation(5ms, 10ms); succ!; };
  **};
end Node.i;

process network
end network;

process implementation network.i
  subcomponents
     s: thread Start.i;
     n0: thread Node.i; n1: thread Node.i; n2: thread Node.i;
  connections
    event port s.start0 -> n0.start; event port s.start1 -> n1.start;
    event port s.start2 -> n2.start; event port n0.succ -> n1.prev;
    event port n1.succ -> n2.prev; event port n2.succ -> n0.prev;
end network.i;

system root
end root;

system implementation root.i
subcomponents
 p: process network.i;
end root.i;
```

Listing 1.1. A token ring in AADL

to support the expression of the semantics of real time preemptible systems even if back-end model checkers do not take into account all the aspects of the model. The tools are implemented and/or integrated in the Topcased environment.

Topcased offers the metamodel of the high-level modeling languages together with graphical editors. The abstract syntax of the Fiacre intermediate language is defined through its metamodel. Transformation languages are used to generate Fiacre models from modeling languages. At this step, some semantics choice must be performed to identify relevant subsets of sources languages and reduce the complexity of intermediate models. Then, source code generators are used to produce the text representation of the Fiacre model and communicate with the external Fiacre front-end. The Fiacre tool performs static analysis of its entry and generates Tina and CADP models, which are analyzed by the corresponding tools.

3.1 The Fiacre Language

Fiacre offers a formal representation of both the behavioral and timing aspects of systems for formal verification and simulation purposes. The design of the language is inspired from decades of research on concurrency theory and real-time systems theory. For instance, its timing primitives are borrowed from Time Petri nets [10], while the integration of time constraints and priorities into the language can be traced to the BIP framework [2]. For what concerns the compositionality of the language, Fiacre incorporates a parallel composition operator and a notion of gate typing which were previously adopted in E-Lotos and Lotos-NT.

Fiacre programs are stratified in two main notions: *processes*, which describes the behavior of sequential components and *components*, which describes a system as a composition of processes, possibly in a hierarchical manner. Listing 1.2 gives an example of a Fiacre program for the token ring example described in Sect. 2.2.

Fiacre is a strongly typed language, meaning that type annotations are exploited in order to guarantee the absence of unchecked run-time type errors. A program is a sequence of declarations. A process is defined by a set of control states and parameters, each associated with a set of *complex transitions*, which are programs specifying how parameters are updated and which transitions may fire. For example, the process declaration:

process T[p : **bool**, q : **none**](v : **int**, &u : **array** 5 **of bool**) **is** ...

expresses that T is a process that may interact over two ports: p, which transmits boolean values, and q, which can only be used for synchronization. The processT has two parameters: v, which is an integer, and u, which is a (reference to a) shared variable.

Complex transitions are built from expressions and deterministic constructs available in classical programming languages (assignments, conditionals, while loops and sequential compositions), nondeterministic constructs (nondeterministic choice and assignments) and communication events on ports. For example, the transition:

from s0 **select** (p!5 ; **to** s1) [] (v :=v + 1 ; **to** s2) **end**

expresses that, in state s0, the process may choose nondeterministically between two alternatives. Either send the value true over the port p and move to state s1, or in-

```
process Start[start0 : none, start1 : none, start2 : none] is
        states s0, s1
        from s0 select start0 [] start1 [] start2 end ; to s1

process Node[prev : none, succ : none, start : in none] is
        states idle, wait, cs, st_1
        from idle select start ; to st_1 [] prev ; to st_1 end
        from st_1 succ ; select to idle [] to wait end
        from wait prev ; to cs
        from cs   succ ; to idle

component root is
        port s0 : none, s1 : none, s2 : none,
             p0 : none, p1 : none, p2 : none,
        par * in
           Start[s0,s1,s2]
         || Node[p0, p1, s0]
         || Node[p1, p2, s1]
         || Node[p2, p0, s2]
        end

root
```

Listing 1.2. A token ring in Fiacre

crement the value of the variable `v` and move to `s2`. A process definition may declare several transitions for the same state. Each can equally be fired.

A component is defined as the parallel composition of processes and/or other components, expressed with the operator **`par`** ... || ... **`end`**. While components are the unit of composition, they are also the unit for process instantiation and for ports and shared variables creation. The syntax of components allows to restrict the access mode and visibility of shared variables and ports, to associate timing constraints with communications and to define priority between communication events. For example, in a component `C`, the declaration **`port`** `p` : **`none`** defines a port called `p` that is private to (cannot be used outside of) `C`. The declaration **`port`** `p` : **`none in`** [`min`,`max`] defines a port that can only interact `min` time units after it has been activated and must be used or deactivated before `max` time units (`min` and `max` should be float or integer constants).

3.2 Translation Principles

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

In the particular instance studied in this paper we illustrate this framework with the translation of AADL models to Fiacre.

The transformation is implemented in the Topcased environment using a model to model transformation approach performed using Kermeta. Kermeta is a model transformation language that combines features coming from Eiffel, Java and OCL. Moreover,

it offers an Aspect Oriented Programming style that makes easy the access to EMF model repositories.

Two main features are heavily used in the translation: high-level iterators on lists and aspect oriented annotations. These annotations allow to specify extensions of existing classes with new attributes and operations. Attributes are used to memorize the Fiacre objects resulting from the transformation of the AADL objects and avoid the use of external data structures such as hash tables.

3.3 Structure of the Generated Code

The code generator has a flatten view of the AADL model as a set of communicating threads. Thus, it associates a Fiacre process to each AADL thread. They do not communicate directly: a *glue* process manages communication and scheduling protocols.

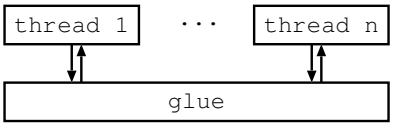

Fig. 2. Threads and the glue

Threads communicate with their environment through the glue process. A thread receives from the glue the values of its input ports and sends to the glue its output events or data ports at specific times. In a first approximation, the glue sends a `dispatch` message to a thread at its logical dispatch time. It takes as parameters the values of the thread input data ports and the value of the triggering event (data) port if any. If the thread is periodic, the value of all the input ports is transmitted. During execution, the thread sends to the glue events with their potential value. When the execution completes, the thread sends a *complete* message to the glue with the values of the output data ports and of the modified (yet not already sent) event data ports.

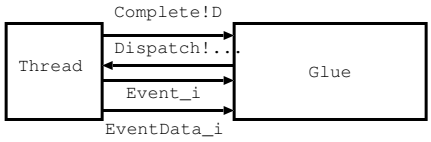

Fig. 3. Communication with the glue

The following code specifies an AADL thread having event, data and event data input or output ports and a data access feature (see Fig. 4).

```
thread T
features
  idp: in data port Tidp;
  odp: out data port Todp;
  iep: in event port {Queue_Size => Q};
  oep: out event port;
  iedp: in event data port Tiedp {Queue_Size => Q};
  oedp: out event data port Toedp;
  m: requires data access Tm;
end T;
```

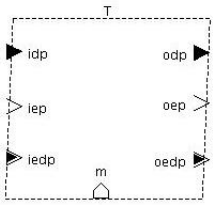

Fig. 4. Thread interface

The interface of the corresponding Fiacre process depends on the way the thread is triggered. If it is timed triggered, the contents of all its input ports is transfered while if it is event triggered, only the contents of the triggering port is transfered:

```
process T[dispatch: in ...,
          complete: out ...,
          oep_port: out none,
          oedp_port: out Toedp]
         (&tab_Tm: array N of Tm, &m: 0..N-1)
  ...
```

Listing 1.3. Translation of time triggered threads

```
type T_events is union C_iep of 0..Q
                     | C_iedp of queue Q of Tiedp
                end
process T[dispatch:in T_events # ...,
    complete: out ...,
    oep_port: out none,
    oedp_port: out Toedp]
    (&tab_Tm: array N of Tm, &m: 0..N-1)
  ...
```

Listing 1.4. Translation of event triggered threads

The translation of thread scheduling is based on Fiacre temporized ports and port priorities. Such ports are used to manage periodic dispatch of threads and non deterministic execution times. Since Fiacre does not offer any support for preemption yet,

we only consider a non preemptive fixed priority scheduler. For the Fiacre translation to remain simple, priorities cannot depend on AADL modes. Only the set of active threads can be mode dependent.

The scheduler manages periodic, sporadic and background tasks. It must ensure data access synchronization as described by the AADL execution model. The following events are managed in our translation.

Dispatch. Dispatch occurs at a multiple of the period (for periodic thread), when a triggering event arrives (for sporadic threads), or at system startup (for background threads). On that event, data are transferred to the thread port variables. A thread can run if all threads that must dispatch at the same time have their data.

Execution. The scheduler allows thread execution. Data received through an immediate connector are transmitted to the thread and thread priorities are encoded using priorities between execution ports.

Completion. The thread ends its execution and transmits its output data connected via immediate connectors and event data not already transmitted.

Deadline. The thread transmits delayed data. Completion must occur before deadline otherwise a schedule error happens.

We have only presented the interface of the main components. Due to the lack of space, we do not describe the implementation and the related state machines. An important aspect of the implementation is the interaction between user threads and AADL execution model. This will be detailed in a forthcoming paper.

3.4 The Considered Subset

Basic properties are considered when generating a Fiacre model. More particularly, (1) AADL modes and priorities are taken into account, as well as (2) access to shared variables.

For the moment, while periods can change, we assume that priorities are fixed. We take into account that connections are determined by the current mode. On the other hand, there is currently no support for multiprocessor architecture in our translation from AADL to Fiacre. As a result, we do not take into account the value of the Actual_Processor_Binding property. We also do not handle preemption. This last feature will be added in a forthcoming version of the Fiacre language.

4 Behavioral Verification with Tina

Tina [11], the TIme Petri Net Analyzer, provides a software environment to edit and analyze Petri Nets and Time Petri Nets. It is particularly well suited to the verification of systems subject to real time constraints, such as those modeled using AADL.

Beside the usual analysis facilities of similar environments, the essential components of the Tina toolbox are state space abstraction methods and model checking tools that can be used for the behavioral verification of systems. This is in contrast with the broader notion of functional verification, in that we attempt to use formal techniques to prove that requirements are met, or that certain undesired behaviors cannot occur —

like for instance deadlocks — without resorting to actual tests on the system. The approach followed here is that commonly referred to as *model-checking*, which basically consists in two abstract steps: (1) the generation of a formal model from a description of the system, followed by (2) a systematic exploration of the states space of this model. This involves exploring states and transitions in the model, relying on smart abstraction techniques to reduce the number and size of these states and therefore reducing the computing time.

The properties to be verified are often described in temporal logics, such as linear temporal logic (LTL) or computational tree logic (CTL). We give some examples of LTL properties related to our running example in Section 4.2.

The result of the verification may lead to an accepting status, meaning that the model of the system satisfies the requirements, or exhibit an error. In the last case, it is often possible to extract a counterexample, which is an explanation at the level of the model (generally an execution trace), which leads to a problematic state. Such counterexamples could be stored alongside an AADL model.

4.1 The Tina Toolbox

The functional architecture of Tina is shown in Fig. 5. The core of the Tina toolset is an exploration engine used to generate state space abstractions that are fed to dedicated model checking and transition system analyzer tools.

The front-ends to the exploration engine convert models into an internal representation — the abstract Timed Transition Systems (TTS) — that is an extension of Time Petri nets handling data and priorities. The *frac* compiler, which converts Fiacre description into TTS and is part of the Topcased environment, is an example of such front-end.

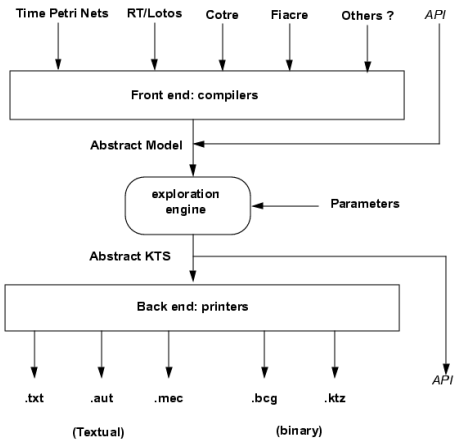

Fig. 5. Tina Architecture

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

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

4.2 Verification

The Tina toolbox provides a native model checker, *selt*, which allows to check more specific properties than the generic properties (boundedness, deadlocks, liveness) that may directly be checked during state space generation. This tool implements an extension of linear time temporal logic known as State/Event LTL [15], a logic supporting both state and transition properties. The modeling framework consists of Kripke transition systems (corresponding to the state class graph of a Petri net in our case), which are directed graphs in which states are labelled with atomic propositions and transitions are labelled with actions. State/Event-LTL formulas are interpreted over the computation paths of the model and may express a wide range of state and/or transition properties.

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

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

We can define some examples or formulas to be checked against the system obtained from our running examples. For instance, the formula `-<>(cs_Node_1 /\ cs_Node_2)` states that it is not the case that, eventually,

the first two processes in the token ring are in critical section (state `cs`) at the same time. Formula (1), which states that at most one `Node` process may be in the critical section at any given moment, offers a more versatile way for expressing mutual exclusion.

```
[](cs_Node_1 + cs_Node_2 + cs_Node_3 <= 1)                    (1)
```

Likewise, we can express that a node in the token ring cannot wait eternally before accessing the critical section. For example, Formula (2) states that "it is always the case that if `Node` waits then, eventually, it enters state `cs`."

```
[](wait_Node_1 => <> cs_Node_1)                               (2)
```

Formulas (1) and (2) can be evaluated (and are true) on the token ring example of Listing 1.2 and on the Fiacre program obtained from the translation of the AADL program in Listing 1.1.

Realtime properties, like those expressed in so-called timed temporal logics, are checked using the standard technique of observers, encoding such properties into reachability properties. The technique is applicable to a large class of realtime properties and can be used to analyze most of the "timeliness" requirements found in practice.

5 Related Work

A number of studies have explored how to interpret the AADL standard in a formal setting. A specification of the AADL execution model in the Temporal Logic of Actions (TLA) is given in [16] that defines one of the earliest formal semantics for AADL. This encoding takes into account a fixed priority scheduling protocol with preemption, the management of modes and communication through ports and shared data. Our approach is based on an interpretation of AADL specifications, including the behavior annex, in the Fiacre Language, which is one of the input languages of the Tina toolbox. A direct encoding from AADL to Petri net is studied in [23] that takes into account a more limited subset of AADL (it restricts the behavior of software components and omits realtime properties of elements). Other target formalisms have also been studied. An encoding of AADL in BIP is presented in [3] that focuses on the behavioral annex as well as on threads, processes and processors. The approach is improved in [14] by taking into account the management of AADL communication protocols. When compared to BIP, the current version of Fiacre provides less high-level constructs – therefore encodings are less direct – but offers better compositional and real-time properties. An interesting study would be to define an intermediate language.

In our work, the behavior of software components can be described using the behavior annex [4], which is currently being defined by the SAE committee. In [6], the authors study the case where behaviors are described in a synchronous language, such as Scade or Lustre. In this case, they define a direct translation that generate an executable model of the software behavior, deployed on the architecture, from an AADL specification. Such a model is usable for early simulation, but also for formal verification,

using tools available for Scade and Lustre. Finally, other works [21,22] have focused on AADL data communication handling but leave the connection with a formal verification tool as a perspective. While we focus our attention on the use of the Fiacre intermediate language and the Tina verification toolset, our work relies also significantly on the Model-Driven Architecture approach promoted in Topcased. Currently, a metamodel for AADL is provided by the OSATE [24] tool, which is integrated in Topcased. A metamodel for Fiacre, build using the Topcased environment, is also available.

6 Conclusion and Future Work

This paper describes a formal verification toolchain for AADL that is currently made available in the Topcased environment. We give a high-level view of the tools involved and illustrate the successive transformations required by our verification process.

Work is still ongoing to improve the tools involved in our verification toolchain. A number of extensions to Tina are being evaluated, concerning new tools, new front-ends, and new back-ends. For instance, we are experimenting with the addition of suspension/resumption of actions to Time Petri nets, which is of great value for modeling scheduled real-time systems. Alongside these works on tools, our current efforts are directed toward three main objectives:

(1) *Simplifying the definition of logical properties.* End users of verification tools should not be required to master temporal logic. To improve the usability of our approach, we are currently investigating the proposition of a kit of predefined AADL requirements. This kit will enable expressing general properties of an AADL component — absence of deadlock, absence of divergence, ... — in a straightforward way.

(2) *Improving error reporting.* We plan to provide a "debugging" procedure, which should take as input a counter-example produced during the model-checking stage and convert it to a trace model of the initial AADL description. These traces should be played back using simulation tools.

(3) *Improving the Verification Process.* We are currently investigating extensions to the Fiacre language in order to ease the interpretation of high-level description languages and to optimize the verification process. One envisioned addition would be to integrate the notion of modes [17] — which is found in a number of ADL, like Giotto and AADL — directly in Fiacre. We also plan to address the problem of specifying scheduling and time-constrained behaviors within Fiacre. These aspects should have a great impact on the overall performance of the analysis tool.

References

1. SAE Aerospace. Architecture Analysis & Design Language (AADL). AS-5506, SAE International, 2004.
2. A. Basu, M. Bozga, and J. Sifakis.Modeling heterogeneous real-time systems in BIP.In *Proc. of SEFM – IEEE Software Engineering and Formal Methods*, 2006.
3. M. Chkouri, A. Robert, M. Bozga, J. Sifakis.Translating AADL into BIP – application to the verification of real-time systems.In *Proc. of MoDELS ACES-MB – Model Based Architecting and Construction of Embedded Systems*, 2008.

4. R. B. Franca, J.-P. Bodeveix, D. Chemouil, M. Filali, D. Thomas, J.-F. Rolland. The AADL behaviour annex, experiments and roadmap. In *Proc. of ICECCS – IEEE International Conference on Engineering of Complex Computer Systems*, 2007.
5. P.-A. Muller, F. Fleurey, D. Vojtisek, Zoé Drey, Damien Pollet, F. Fondement, P. Studer, and J.-M. Jézéuel. On executable meta-languages applied to model transformations. In *Proc. of MoDELS – Model Transformations In Practice*, 2005.
6. E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, D. Lesens. Virtual Execution of AADL Models via a Translation into Synchronous Programs. In *Proc. of EMSOFT – ACM & IEEE international conference on Embedded software*, 2007.
7. F. Jouault, and I. Kurtev. Transforming Models with ATL. In *Proc. of MoDELS – Model Transformations in Practice*, 2005.
8. OAW, `http://www.openarchitectureware.org/`.
9. OCL, UML 2.0 Object Constraint Language.
10. P. M. Merlin and D. J. Farber. Recoverability of communication protocols: Implications of a theoretical study. *IIEEE Transactions on Computers*, 24(9):1036–1043, 1976.
11. B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research*, 42(14), 2004.
12. H. Garavel, F. Lang, R. Mateescu, W. Serve. CADP: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. of CAV – Int. Conf. On Computer Aided Verification*, 2007.
13. B. Berthomieu, J. P. Bodeveix, M. Filali, H. Garavel, F. Lang, F. Peres, R. Saad, J. Stoecker, F. Vernadat. The syntax and semantics of Fiacre. Research Report LAAS 07264, 2007.
14. L. Pi, J.-P. Bodeveix, M. Filali. Modeling AADL Data Communication with BIP. Preprint, 2009.
15. S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, N. Sinha. State/Event-based Software Model Checking. In *Proc. of IFM – Integrated Formal Methods*, LNCS Vol. 2999, 2004.
16. J.-F. Rolland, J.-P. Bodeveix, D. Chemouil, M. Filali, D. Thomas. Towards a formal semantics for AADL execution model. In *Proc. of ERTS – European Congress on Embedded Real-Time Software*, 2008.
17. J.-F. Rolland, J.-P. Bodeveix, M. Filali, D. Thomas, D. Chemouil. Modes in asynchronous systems. In *Proc. of UML&AADL*, 2008.
18. Topcased: "Toolkit in OPen-source for Critical Apllications and SystEms Development", `http://www.topcased.org`.
19. B. Berthomieu, and F. Vernadat. State Space Abstractions for Time Petri Nets. Handbook of Real-Time and Embedded Systems, Chapman & Hall, 2007.
20. J.-M. Farines, B. Berthomieu, J.-P. Bodeveix, P. Dissaux, P. Farail, M. Filali, P. Gaufillet, H. Hafidi, J.-L. Lambert, P. Michel, and F. Vernadat. The Cotre Project: Rigorous Software Development for Real Time Systems in Avionics. In *Proc. of FMICS – Formal Methods for Industrial Critical Systems*, vol. 80 of ENTCS, 2003.
21. C. André, F. Mallet, R. de Simone. Modeling of immediate vs. delayed data communications: from AADL to UML Marte. In *Forum on specification & Design Languages*, 2007.
22. P. Feiler. Efficient embedded runtime systems through port communication optimization. in *Proc. of ICECCS – IEEE International Conference on Engineering of Complex Computer Systems*, 2008.
23. T. Vergnaud. Modélisation des systèmes temps-réel répartis embarqués pour la génération automatique d'applications formellement vérifiées. PhD Thesis, École nationale supérieure des télécommunications, 2006.
24. The SEI AADL Team. *An Extensible Open Source AADL Tool Environment (OSATE)*. Software Engineering Institute, 2006.