Behavioural descriptions in architecture description languages Application to AADL

Jean-Paul Bodeveix, P Dissaux, M Filali, P Farail, P Gaufillet, François Vernadat

▶ **To cite this version:**

Jean-Paul Bodeveix, P Dissaux, M Filali, P Farail, P Gaufillet, et al.. Behavioural descriptions in architecture description languages Application to AADL. 3rd European Congress on Embedded Real Time Software (ERTS 2006), Jan 2006, Toulouse, France. hal-02270356

HAL Id: hal-02270356
https://hal.science/hal-02270356

Submitted on 24 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Behavioural descriptions in architecture description languages Application to AADL

J.-P. Bodeveix[2], P. Dissaux[3], M. Filali[2], P. Farail[1], P. Gaufillet[1], F. Vernadat[2],
1. AIRBUS France pierre.gaufillet@airbus.com
2. FéRIA {bodeveix,filali}@irit.fr francois@laas.fr
3. TNI-World pierre.dissaux@tni-world.com

November 10, 2005

1 Introduction

The development of critical software has put forward architecture description languages. The aim of these languages is to allow a verification process at the early stages of software development. They cover many aspects like real time, information flow, distribution, execution platforms, ...

The Architecture Analysis and Design Language (AADL)[8] standard was prepared by the SAE AS-2C Architecture Description Language Subcommittee, Embedded Computing Systems Committee, Aerospace Avionics Systems Division. The AADL standard is based on MetaH[9], an architecture description language developed at Honeywell Laboratories under the sponsorship of the US Defense Advanced Research Projects Agency (DARPA) and US Army Aviation and Missile Command (AMCOM). Release 1.0 of the AADL standard (SAE AS5506) has been issued in November 2004.

The AADL is a language used to describe the software and hardware components of a system and the interfaces between those components. The language can describe functional interfaces of components, such as data inputs and outputs, and 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. More detailed information about this language may be found at: www.aadl.info.

Although some behavioural aspects can be described in AADL (e.g. static linear control flows), application behaviours rely mainly on source code written in a target language (C, Ada, ...). This paper presents a behaviour extension as properties and a dedicated annex in order to make possible a better analysis of behaviours. These extensions try to reuse as much as possible existing AADL concepts. The main features of our proposal concern:

- High level composition paradigms: we have studied how to introduce HRT-HOOD [4] composition paradigms within AADL.
- Dynamic aspects related to subprogram and thread behaviors. We have adopted an automata based description similar to AADL mode transitions.

2 Behaviour descriptions for AADL

This proposal[1] introduces some HRT-HOOD concepts within AADL and attach behaviours to subprograms and threads. HRT-HOOD concepts are specified through dedicated AADL properties. A behavioural annex extending AADL mode automata is defined to specify subprogram and thread behaviours. In the following, we present the behavioral annex attached to AADL components and how synchronization mechanisms existing in HRT-HOOD can be specified in the AADL framework. Lastly, we extend the behavioral specification to take into account real-time aspects.

2.1 Subprogram and thread behaviour

An AADL mode automaton is defined with a set of modes with an initial mode and mode transitions enabled by events, using the syntax:

```
m1 -[ event ]-> m2
```

In the proposed annex, we use the same syntax to define state transitions. However, an action is attached to a transition and activation condition can be specified by an event, a predicate or both. Furthermore, besides the *initial* keyword of AADL modes, we introduce a *return* keyword for subprogram implementation states. When reached, control returns to the caller. However, the automaton can be non deterministic and may specify several possible effective implementations. Moreover, within an automaton, we can access the subprogram parameters according

[1] In fact, this proposal has been elaborated from the one[7] already done in the context of AADL 0.95.

to their declared mode (in or out). The following text illustrates the attachment of an automaton to a subprogram.

```
subprogram addition
  x: in parameter std::integer;
  y: in parameter std::integer;
  r: out parameter std::integer;
  ovf: out parameter std::integer;
end addition;

subprogram implementation addition.default
annex cotre {**
  states
    s0 : initial state;
    s1 : return state;
  transitions
    s0 -[ ]-> s1 { r := x + y ; ovf := false; }
    s0 -[ ]-> s1 { r := 0; ovf := true; }
**};
end addition.default;
```

Whereas AADL flows denote sequences of calls, the proposed behavioural annex allows the expression of control flows which take into account state variables and parameters.

2.1.1 Thread behaviour

The behavior of a thread can be specified using the same annex. However, the thread terminates if a *final* state is reached. Termination is not mandatory: the thread can be active forever.

2.1.2 Subprogram invocation

The AADL standard supports the declaration of subprogram call sequences which may be conditionned by a mode. Calls are identified by an occurrence. The occurrence identifier is used to specify data flows between in and out parameters of called subprograms. For example, consider the `addition` subprogram which may raise an event if an overflow occurs:

```
subprogram addition
features
  x: in parameter std::integer;
  y: in parameter std::integer;
  r: out parameter std::integer;
  ovf: out event;
end addition;
```

The following `test` subprogram performs two calls to the `addition` subprogram. The implementation of the subprogram contains two calls to `addition`, identified by `add1` and `add2`. The `connexion` section specifies how data flows from local variables to in parameters of the first call and then to in parameters of the second call:

```
subprogram test
features
  ovf: out event;
end test;

subprogram implementation test.default
subcomponents
  x: data Cotre::integer;
  y: data Cotre::integer;
  z1: data Cotre::integer;
  z2: data Cotre::integer;
-- AADL standard:
-- specification of call occurrences
calls {
  add1: subprogram addition;
  add2: subprogram addition;
};
-- fixed data flow graph
connections
  parameter x -> add1.x;
  parameter y -> add1.y;
  parameter add1.z -> z1;
  parameter z1 -> add2.x;
  parameter y -> add2.y;
  parameter add2.z2 -> z;
  event add1.ovf -> ovf;
  event add2.ovf -> ovf;
end test.default;
```

However, such a specification is very restrictive. The order in which calls are performed is fixed and cannot be data dependent. Different call sequences can be associated to different modes. The annex we propose can specify data dependent behaviors. The implementation of a subprogram can be specified as performing other subprogram calls or event notifications. The same syntax is used for both: `p!x1,...,xn` express the call to subprogram `p` with actual parameters `x1`,...`xn` or the sending of an event to the port `p`. If `p` is a data port or port group, the values `x1`,...,`xn` are transmitted:

```
annex cotre_behavior {**
  states
    s0: initial state;
    s1: state;
```

```
    s2: final state;
  transitions
    s0 -[addition!x,y,z1]-> s1 {}
    s1 -[when z1 < 10 &
         addition!z1,y,z2]-> s2 {}
    s1 -[when z1 >= 10]-> s2 { z2 := z1; }
**};
```

Remark Operation calls and event notifications can also appear in the action associated with a transition. In this case, we will write:

```
annex cotre_behavior {**
  states
    s0: initial state;
    s1: state;
    s2: final state;
  transitions
    s0 -[ ]-> s1 { addition!x,y,z1; }
    s1 -[when z1 < 10]-> s2
       { addition!z1,y,z2; }
    s1 -[when z1 >= 10]-> s2 { z2 := z1; }
**};
```

2.1.3 Sendind and receiving messages

Messages are sent and received through AADL ports or groups of ports according to a syntax which is similar to that of subprogram calls. The **!** symbol used to indicate a subprogram call also indicates that a message has been sent when the identifier is a port or group of ports. The parameters are data sent through the port or through each port in the group. Parentheses can be used to separate the sub-groups of a hierarchical group of ports. The **?** symbol indicates that a message is received from a port or a group of ports.

```
thread test
features
  p_in: in event data port Cotre::integer;
  p_out: out event data port Cotre::integer;
end test;

thread implementation test.default
subcomponents
  x: data Cotre::integer;
annex cotre_behavior {**
  states
    s0: initial state;
    s1: state;
  transitions
    s0 -[p_in?x]-> s1 {}
    s1 -[p_out!x+1]-> s0 {}
**};
end test.default;
```

Remark As for operation calls, sent / received messages can also appear in the action part, but the semantics are different: a transition is selected from among those which can be immediately fired. If a synchronization is in the action part, the transition waits for the synchronization whereas if the synchronization is in the event part of the transition, the firing of the transition is conditionned by the selection of this transition among all the enabled transitions.

2.2 Passive objects

Passive objects are translated directly to AADL data components. A data declaration corresponds to a class declaration encapsulating data and exporting access methods. However, the profile of methods must be declared outside the component via a subprogram component type. The data component and the subprograms defining the method profiles can be grouped together in a package.

```
subprogram put
features
  v: in parameter Cotre::integer;
end put;

subprogram get
features
  v: out parameter Cotre::integer;
end get;

subprogram empty
features
  v: out parameter Cotre::boolean;
end empty;

subprogram full
features
  v: out parameter Cotre::boolean;
end full;

data stack
features
  put: subprogram put;
  get: subprogram get;
  empty: subprogram empty;
  full: subprogram full;
```

```
end stack;
```

A behavior can be associated with these subprograms in an annex. However, this behavior would need to access to the implementation of the data structure, which would require either adding a new parameter (a reference to the current instance), either an AADL connection between each subprogram and the data. It seems better to include the subprogram behaviors in the data structure implementation. For this purpose, the annex must declare several entry points. Initial states are attached to each subprogram[2].

```
data implementation stack.default
  elems: data Cotre::integer
     { Multiplicity => 10; };
  sp: data Cotre::integer;

annex cotre_behavior {**
inits
  sp := 0;
states
  g : initial state for get;
  p : initial state for put;
  e : initial state for empty;
  f : initial state for full;
  s1 : return state;
transitions
  g -[when sp > 0]->s1
     { sp := sp - 1; v := elems[sp]; }
  p -[when sp < 10]->s1
     { elems[sp] := v; sp := sp + 1; }
  e -[]->s1 { v := (sp = 0); }
  f -[]->s1 { v := (sp = 10); }
**};
end stack.default;
```

A single annex can contain the behavior for several subprograms via several initial state declarations. The behavior of several subprograms can also be distributed in different annexes. The action part of a transition can call a subprogram of the current data or of another data.

2.3 Shared objects

AADL supports the notion of protected object: a protected object is a data component whose Concurrency_Control_Protocol property is positioned at protected. The access control mechanisms are actually left open by AADL and must be defined via the Supported_Concurrency_Control_Protocols property.

Taking the example of the stack, we will add a property to the specification:

```
data stack
features
  put: subprogram put;
  get: subprogram get;
  empty: subprogram empty;
  full: subprogram full;
properties
  Concurrency_Control_Protocol => protected
end stack;
```

2.4 Active objects

As in HRT-HOOD, the proposed modelization of active objects allows the separation of synchronization from computation. Active objects specify several entry points of which behavior is essentially sequencial, as described in the previous paragraph. Synchronization modes are expressed via events or the AADL client-server mode. However, these notions are not sufficient to express activation conditions. To do this, we attach a behavior to the thread of the active object. This behavior is also described by an automaton and expresses the activation conditions. Consequently, one or several annexes of an active object describe the behavior of the entry points and the behavior of the active object itself. The communication between the clients, the object thread and the entry points is specified by the synchronization mode attached to the entry point. Three modes taken from HRT-HOOD are proposed and extend AADL client server protocol:

- ASER: the invocation is non blocking. These invocations are stored in a queue which can be bounded through the attribute `Queue_Size`.
- HSER: the invocation is blocking. The caller resumes once the call returns.
- LSER: the invocation is blocking. The caller resumes once the call is accepted.

For such purposes, we have introduced an AADL property declared as follows:

```
Server_Call_Protocol : type enumeration
  (ASER,HSER,LSER) → HSER
    applies to (server subprogram);
```

[2]We reuse the `multiplicity` property of UML to specify arrays.

Remark: HSER is the implicit property attached to subprograms. Actually, it is the implicit subprogram call protocol in AADL.

2.4.1 Asynchronous client-server mode

In the asynchronous mode several clients can invoke the services of a server object. The invocation is non-blocking. The requested processing is carried out in parallel with the client. The entry points are represented by ports which may support data. These events are stored in files that can be bounded via the Queue_Size attribute. The activation conditions are defined by the behavior of the thread associated with the active object. This thread can accept the events issued by the clients and may call local subprograms which perform the requested processing.

For example, the sender thread of the alternated bit protocol can be specified in AADL extended by the cotre behavioral annex as follows: the thread specification declares two input event ports receiving acknowledgements of each sign, two ouput data ports transmitting data to the receiver, each port being associated with a sign, and one input data port to get information to be transmitted.

```
thread sender
features
  -- data to be transmitted
  put: in event data port;
  ack0: in event port; -- positive ack
  ack1: in event port; -- negative ack
  -- data transmission with negative tag
  put0: out event data port;
  -- data transmission with positive tag
  put1: out event data port;
end sender;
```

The implementation sends data with a given sign until it receives an acknowledgement of the same sign. As either data or acknowledgement can be lost, data must be sent again. A timeout should be specified in order to avoid to frequent data sendings. When the acknowledgement has been received a new value is obtained from the put port and transmitted through ports associated to the opposite sign.

```
thread implementation sender.default
  v: data;
annex cotre_behavior {**
states
  s0: initial state;
  s01, s02, s1, s11, s12: state;
transitions
  s0 -[put?v]-> s01 {}
  s01 -[put0!v]-> s02 {}
  s02 -[ack0?]-> s1 {}
  s02 -[]-> s01 {} -- timeout
  s1 -[put?v]-> s11 {}
  s11 -[put1!v]-> s12 {}
  s12 -[ack1?]-> s0 {}
  s12 -[]-> s11 {} -- timeout
**}
end sender.default;
```

We propose here to extend the client/server subprogram feature of AADL so that it supports asynchronous communications. The ASER mode must be attached to asynchronous entry points. The sender thread of the alternated bit protocol is then declared as follows. Input ports become server subprograms while output ports are declared as required subprograms: they will be connected to entry points of the communication channel.

```
thread sender
features
  put: server subprogram put
    { Server_Call_Protocol => ASER; };
  ack0: server subprogram ack
   { Server_Call_Protocol => ASER; };
  ack1: server subprogram ack
   { Server_Call_Protocol => ASER; };
  put0: requires subprogram put;
  put1: requires subprogram put;
end sender;
```

The implementation remains unchanged as message sending and subprogram calls have the same syntax. It has to be noted that the subprograms declared here do not need to be implemented. All the processing is performed by the thread.

2.4.2 Synchronous client-server mode

The HSER mode of HRT-HOOD enables the service which has been called up to return a result to the client. The corresponding synchronization is the client-server mode of AADL. Remember that the client-server mode of AADL is not able to take account of activation conditions.

The proposed solution involves using the object thread to express the acceptance conditions of the requests sent to each server subprogram. The operation activation constraints can be expressed via a 3-way synchronization where the client and the server must

be ready to synchronize and execute the subprogram code.

In the example below, it is possible to call the put operation (resp: get) if the buffer is empty (resp. full). These conditions are expressed by the server's behavior automaton. The AADL client-server synchronization is performed after the condition has been checked.

The semantics of the component behavior is therefore as follows: a thread is created and executes the code from the initial state. The thread is suspended on a pending event which corresponds to an entry point. Because the client/server communication mode is synchronous, the client is suspended until completion. Completion occurs when the called subprogram reaches a return state.

```
thread serveur
features
  put: server subprogram buffer.put;
  get: server subprogram buffer.get;
end serveur;

thread implementation serveur.default
subcomponents
 buf: data buffer;
annex cotre_behavior {**
states
 empty: initial state;
 p : initial state for put;
 g : initial state for get;
 r : return state;
 full : state;
transitions
 -- expression of acceptance condition
 empty -[put?v]-> full { };
 full -[get?v]-> empty { };
 -- code for entry points
 p -[]-> r { buf.put!v; }
 g -[]-> r { buf.get!v; }
**}
end serveur.default;
```

Remarks It may be possible to express the acceptance conditions concerning the subprogram parameters. The example below only accepts put calls with a positive argument. A call with a negative or null argument blocks the client.

```
annex cotre_behavior {**
states
 full : state;
 empty : initial state;
transitions
 empty -[ put ?x & x > 0 ]-> full { };
 full -[ get ?]-> empty { };
**}
```

2.4.3 Semi-Synchronous client-server mode

In the HRT-HOOD semi-synchronous mode, the client is unblocked when the message is received. In this mode, only the input parameters are transmitted. The server thread synchronizes with the message sent by the client and calls a local subprogram which processes the request. The synchronization (performed by the thread associated with the server) and the computation (performed by the subprogram) are always separate.

The semi-synchronous and synchronous modes are distinguished by the client wakeup time (which occurs when the request is accepted for the semi-synchronous mode and at the end of processing for the synchronous mode). To do this, we associate respectiveley the LSER and HSER property values to the entry point.

The following example declares the semi-synchronous entry point `put`. Only positive data are stored. Attempting to store negative or null data results in client suspension. The client is woken up before the effective subprogram call by the server thread.

```
thread serveur
features
  put: server subprogram buffer.put;
        { Server_Call_Protocol => LSER; };
end serveur;

thread implementation serveur.default
subcomponents
 buf: data buffer;
annex cotre_behavior {**
states
 s0 : initial state;
 p: initial state for put;
 r: return state;
transitions
 -- client wake up if x > 0
 s0 -[ put?x & x > 0]-> s0 { };
 p -[]-> r { buf.put!x };
**}
end serveur.default;
```

2.4.4 Combining synchronization modes

The homogeneous declaration of the various synchronization mechanisms makes it easier the specification of the interface of a thread. The following examples describes a monitor allocated read or write access rights to clients. The entry points `start_read` and `start_write` are synchronous: a client must wait for the completion of the call before accessing the ressource. The entry points `end_read` and `end_write` can be asynchronous: the client does not need to wait for the completion of the request.

```
thread RW_monitor
features
  start_read: server subprogram start_read
     {Server_Call_Protocol => HSER; };
  end_read: server subprogram end_read
     {Server_Call_Protocol => ASER; };
  start_write: server subprogram start_write
     {Server_Call_Protocol => HSER; };
  end_write: server subprogram end_write
     {Server_Call_Protocol => ASER; };
end RW_monitor;
```

2.5 Timing aspects

Timing aspects are already present in AADL through properties associated to various components. However, the behavioral extension must also provide implementation related timing constructs.

2.5.1 Elapse of time

Apart from assignments, procedure calls and event notification, it is possible to associate timed actions with a transition:

- Computation(min,max) expresses use of the cpu for a non-deterministic period of time between min and max.
- Delay(min,max) expresses a suspension for a non-deterministic period of time between min and max.

Remark: with respect to the elapse of time, these two actions express the same thing. However, they are meaningful with respect to scheduling purposes.

2.5.2 Periodic threads

A periodic thread is declared using pre-declared AADL attributes: `Dispatch_Protocol=>Periodic` and its period `Period`. The behavior described by the Cotre annex is triggered from its initial state and must reach a final state before the `Compute_Deadline`.

```
thread Emetteur
features
   lput: subprogram put;
end Emetteur;

thread implementation Emetteur.default
properties
 Dispatch_Protocol => Periodic;
  Period => 10 ms;

annexe cotre.behavior {**
states
  s0: initial state;
  s1: final state;
transitions
  s0 -[ lput ! ]-> s1 { };
**}
end Emetteur.default
```

2.5.3 Interaction within a bounded time

Several techniques can be used to bound the waiting time for a synchronization when a subprogram is called, depending on the declaration to which the attribute specifying a time boundary is attached:

- to subprogram declarations, in accordance with the HRT-HOOD syntax,
- to subprogram import declarations,
- to subprogram calls or message dispatch.

In order to simplify the semantics of the bounded wait, a boundary can only be attached to a synchronization on a port or subprogram in LSER mode. If a HSER call could be bounded, we would have a problem with respect to the atomicity of the server computation.

The solution we have chosen to express the bounded wait involves adding a timeout transition which is triggered when the system has remained in a given state for a certain amount of time. Thus, the time is counted down from the moment the client is ready to synchronize.

The timeout is introduced via a logic expression which takes the form g timeout T which is satisfied if g has been true for at least T units of time. Given

that the transitions are fired as soon as possible, a synchronization with timeout is expressed as follows:

```
s0 -[ g & p! ]-> s1  { }
s0 -[ g & timeout T ]-> s2 { }
```

3 Properties specification

In the context of real time or concurrent programming, the properties generally address the dynamic behavior of the system - notions which are harder to capture using only predicate calculus. These properties are more often, and more easily, expressed using temporal logics assertions [5, 1]. In order to help a user - not necessarily a specialist of temporal logics, we propose an interface using so called "domain properties" that we present now. Those "domain properties" are the sets of generic properties a user working in this particular field is most likely to wish to verify. To allow specification of more specific properties, we also considered the introduction of "tool properties" directly written in the specification language of a given verification tool. In this draft, we will only discuss properties of the former kind.

The "domain properties" are organized into two categories:

(1) a set of "general" properties

(2) a set of "component specific" properties

For each property, we give an equivalent one in terms of basic operators.

3.1 General properties

We distinguish two kinds of properties expressing a general behavior of a specific component. Depending on whether a property is located in the root component or in a sub-component, one specifies a global or a local property.

- The first kind is "state oriented" and specifies the possibilities for a component (system) to go back to an initial state.
- The second kind is "event-oriented" and specifies liveness of the component (system) or the possibility of divergence (infinite computation without "observable" event).

Re-initializable

- **resettable_pot** ($\equiv AG\ EF\ initial$)

  The component may eventually return to its initial state (from any state).

- **resettable_inev** ($\equiv AG\ AF\ initial$)

  The component must eventually return to its initial state (from any state).

Component liveness Let C be the subset of events of a component. We extend CTL operators EF and AF to EF_C and AF_C. EF_C specifies that some path from starting from the current state contains a transition of C, AF_C specifies that all paths starting from the current state contains a transition of C.

- **is_alive** ($\equiv AG\ EF_C$ true)

  Always some action of the component must be possible in the future

  When used in the root component, this property implies absence of deadlock.

- **no_livelock** ($\equiv AG\ AF_C$ true)

  A component may not be indefinitely idle (without performing some action)

3.2 Specific properties

In order to specify more specific properties, we propose also a small set of "properties patterns". In the sequel, $e_1, e_2 \ldots$ denote boolean expression on atomic variables characterizing the behavior of the component.

invariant p ($\equiv AG\ p$)
Property p always holds

Leadsto
e_1 **leadsto** e_2 [**within** d] $\equiv AG\ (e_1 \Rightarrow\ AF_{\leq d}\ e_2)$
When expression e_1 holds then expression e_2 will eventually holds [at most in d unit time later].

Reachable from
reachable e_1 [**from** e_2][**within** d]$\equiv\ AG(e_1 \Rightarrow EF_{\leq d}\ e_2)$
When expression e_1 holds then expression e_2 potentially holds in the future [at most in d unit time later].

After
e_1 **after** $e_2 \equiv A[\neg e_2\ Weak\ Until\ e_1]$
When expression e_1 holds then expression e_2 has been necessarily satisfied in the past

Remark: Taking benefit of the presence of labels of transitions it is also possible to envisage the use of behavioral equivalences (bisimulation, testing equivalence, ...) to specify directly by means of an abstract automaton the desired behavior of the considered part of the system (component system).

4 Conclusion

We have considered behavioural aspects within an architecture description language, namely AADL. Our proposition has tried to keep as much as possible close to AADL. We are now investigating how we can define a formal semantics to some of the AADL constructs [3]. Moreover, starting from our experience in the Cotre project [2], for verification purposes, we will also investigate the mapping of AADL constructs to those available within verification tools and more generally how to deal with AADL transformations [6].

References

[1] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(1):183–235, February 1994.

[2] B. Berthomieu, P.-O. Ribet, F. Vernadat, J.-L. Bernartt, J.-M. Farines, J.-P. Bodeveix, M. Filali, G. Padiou, P. Michel, P. Farail, P. Gaufillet, P. Dissaux, and J.-L. Lambert. Towards the verification of real-time systems in avionics: The Cotre approach. In *Eigth International workshop for industrial critical systems, ROROS*, pages 201–216. Thomas Arts, Wan Fokkink, 5-7 juin 2003.

[3] J.-P. Bodeveix, D. Chemouil, M. Filali, and M. Strecker. Towards formalizing AADL in proof assistants. In J. Kuster-Filipe, I. Poernomo, R. Reussner, and S. Shukla, editors, *Formal Foundations of Embedded software and component-based softare architectures (ETAPS), Edinburgh*, pages 137–153. LFCS (University of Edinburgh), 2-10 april 2005.

[4] A. Burns and A. Wellings. *HRT-HOOD a structured design method for hard real-time Ada Systems.* Elsevier, 1995.

[5] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite state concurrent system using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2), 1986.

[6] P. Dissaux. AADL model transformation. In *DASIA*, june 2005.

[7] P. Farail and P. Gaufillet. Cotre as an AADL profile. In *Architecture Description Languages*, pages 167–180. IFIP, Springer, 2004.

[8] P. H. Feiler, B. Lewis, and S. Vestal. The SAE architecture analysis & design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In *RTAS Workshop 2003*, pages 1–10, May 2003.

[9] S. Vestal. *MetaH User's Manual.* Honewell Technology Drive, 1998. http://www.htc.honeywell.com/metah/uguide.pdf.