COTRE AS AN AADL PROFILE

Patrick Farail, Pierre Gauffillet

Airbus France, 316 route de Bayonne, 31060 Toulouse Cedex 03, France

Abstract: The COTRE Architecture Description Language, developed during the COTRE project, allows software designers to describe hierarchically static and real time architectures, their behavior and their functional or non-functional requirements. Due to different needs at the user and at the verification levels, the COTRE language has been split early in the project into 2 specialized sub-languages named respectively UCOTRE (User-COTRE) and VCOTRE (Verification-COTRE). As the UCOTRE concepts are very close to the AADL ones, and to avoid defining 2 concurrent languages, UCOTRE has become an AADL dialect, using its extension mechanisms. This paper focuses on the UCOTRE description as a set of extensions and restrictions to AADL.

Key words: Architecture Description Language, Real Time, Software engineering, model checking, modeling, COTRE, AADL

1. INTRODUCTION

With the increasing size and complexity of embedded systems, it is more and more important to have development methods allowing to keep the products mastery and to ensure working order and safety. This is all the more true for equipment like some avionics computers, and more generally for critical systems (flight commands, engine controller, nuclear domain, etc.). For this kind of product, we have to deal not only with functional requirements, but also with non-functional aspects like time, reliability and performance constraints.

By aiming to provide a design methodology and tools based on dynamic (aka threads, semaphores, buffers, ...) and static (modules, objects, classes, ...) software architecture modeling, the COTRE research project addresses this problem^{1,2}.

Figure 1. The COTRE transformations

Started at the beginning of 2002, the COTRE project has been funded by the French Ministry of Research & Education through the RNTL (Réseau National des Technologies Logicielles – Software Technologies National Network) for 2 years. This project gathers an end user, AIRBUS France³, a software tool editor, TNI-Valiosys⁴, FeRIA⁵ – CNRS LAAS⁶, UPS IRT⁷ and ONERA-DTIM⁸ – and ENSTB⁹ as academic partners specialized in formal languages and verification techniques.

As the ADL (Architecture Description Languages) are used to describe structural decomposition of systems and interactions between their components, it seems natural to express the COTRE models in a such language. The COTRE language, in addition to the common features of ADL, must :

- allow to describe formally the constraints the system must comply with,
- allow to describe formally its behavior,
- and have an associated graphic expression to make easier the system designer job.

To allow to work more easily on the user level modeling and on the verification layers, the abstract COTRE language is divided into two languages : the UCOTRE language, standing for User COTRE, and the VCOTRE language, for Verification COTRE. VCOTRE can be seen as an intermediate language between UCOTRE and the specific formalisms used by the verification tools (see Illustration 1). UCOTRE and VCOTRE should ultimately merge into an unique COTRE language.

The proximity of the UCOTRE needs and the AADL (Architecture Analysis & Design Language, formerly known as Avionics Architecture Description Language) ones leads us to open technical discussions with the AS-2C SAE subcommittee¹⁰ in charge of its development.

In the description of the VCOTRE language¹¹, translations in the chosen verification formalisms (Time Petri Net, Timed Automata, Temporal/Timed Logics) and an application example are presented. Verification techniques which we are implementing in the COTRE platform, are based on formula satisfaction (temporal logic model checking) or on model comparison (behavioral equivalencies). Expressing and checking these properties may require different models, methods and tools, each kind of model typically comes with its tool suite. In the context of COTRE, the following tools have been investigated : UPPAAL¹², SMV¹³, TINA¹⁴, LPV¹⁵.

In this paper, we will focus on the UCOTRE language, and its building based on the AADL. The section 2 deals with the AADL extension mechanisms, and the section 3 exposes, around a complete example already studied at the VCOTRE level¹, the main aspects of the UCOTRE language.

2. THE AADL EXTENSION MECHANISMS

The AADL core, in its version 0.96, offers 2 main extension mechanisms :

- Property sets : each architecture element can be associated statically with named values. The names, as their allowed values and the element types they can be associated to are described thanks to the property set statement.

PROPERTY SET my_ext **IS**

```
int    : TYPE INTEGER -32767..32768;
duration : TYPE UNITS (s, j => s * 86400,
                      h => s * 3600,
                      min => s * 60,
                      ms => s * 0.001,
                      us => s * 0.000001);
```

```
rate : float => 0.0 APPLIES TO (SUBPROGRAM);
END my_ext;
```

In this example, `my_ext::int` and `my_ext::duration` are 2 new property types, and `my_ext::rate` is a new property of type `float`, with a default value of `0.0`, only usable for subprograms.

- **Annexes** : specific sections called annexes can be added to every AADL component. The content of these annexes is entirely defined by the user. If a software tool does not know an annex, it just has to ignore it. In this way, a minimum compatibility level is ensured.

SYSTEM IMPLEMENTATION `ex.default`

ANNEX `<name>` **IS**

`<free syntax>`

END ANNEX `<name>;`

END `ex.default;`

3. THE UCOTRE LANGUAGE

The UCOTRE meta-model was initially derived from the HOOD one (both HOOD 4 and HRT-HOOD¹⁶) and from real time objects used in the AIRBUS development process. When the UCOTRE language started to get closer to AADL¹⁷, some new concepts had to be added, as the AADL types / implementations / instances features for example.

At this moment, some features are not supported in UCOTRE. The most important limitation is the lack of ports : as in the HOOD model, and because UCOTRE currently is not intended to handle multi-applications architectures, components communicate only thanks to subprogram calls. Nevertheless, ports should be used in a later version of UCOTRE to model communications between applications.

The software architecture presented here is called `deadlock_verification`. This example has already been used to illustrate the VCOTRE language¹⁸. In this system, as represented in Illustration 2, two periodic processes `t_1` and `t_2` use 2 semaphores `sem1` and `sem2` in the same or in the inverted order. The verification problem consists in the analysis of the deadlock possibility.

Figure 2. deadlock_verification HOOD view

The architectural description of this system with UCOTRE is :

SYSTEM deadlock_verification

END deadlock_verification;

SYSTEM IMPLEMENTATION deadlock_verification.default

SUBCOMPONENTS

dp : **PROCESS** Partition.A;

ANNEX cotre.guarantees **IS**

IS ALIVE;

END ANNEX cotre.guarantees;

END deadlock_verification.default;

sys_deadlock_instance: **SYSTEM** ex_deadlock.default { };

PROCESS Partition

END Partition;

PROCESS IMPLEMENTATION Partition.A

SUBCOMPONENTS

t_1 : **THREAD** t.t1(sem1 => sem_1, sem2 => sem_2);

t_2 : **THREAD** t.t2(sem1 => sem_2, sem2 => sem_1);

sem_1 : **DATA** semaphore.default;

sem_2 : **DATA** semaphore.default;

END Partition.A;

This system only consists of the application `dp`, which contains the 2 threads `t_1` and `t_2` and the 2 semaphores `sem_1` and `sem_2`. Inside this fragment, you can notice :

- first, the definition of the type of system `deadlock_verification`,
- then, the definition of the implementation of this type,
- and finally the instantiation of the system. This instance corresponds to the HOOD `system_configuration` concept.

At this level, the only COTRE specific construction is the contract `IS ALIVE` inside the `cotre.guarantees` annex, ensuring the system can not reach a deadlock. The UCOTRE contracts have been almost directly re-used from the VCOTRE layer¹. They can express assumptions about component environment by the mean of the `cotre.assumes` annexes, and guarantees about component behavior in the `cotre.guarantees` annexes. It can be done using high level assertions or behavioral equivalences (based on automata).

```
ANNEX cotre.guarantees | cotre.assumes IS
((<assertion> | <behavioral equivalence>);)+
END ANNEX cotre.guarantees | cotre.assumes;
```

The following assertions are defined :

Table 1. Assertions

Assertion	Formal description	Comments
potentially reset	$AG\ EF\ init$	From any state, the component may go back to its initial state.
unavoidably reset	$AG\ AF\ init$	From any state, the component must go back to its initial state.
is alive	$AG\ EF\ EX_c\ true$	Some actions have always to be possible in the future. Applied to a root component, this assertion implies that there is no deadlock.
no livelock	$AG\ AF\ EX_c\ true$	The component must not stay forever idle.
invariant<exp>	$AG\ <exp>$	$<exp>$ has always to be true.
<exp1> leads to <exp2> [within <exp3>]	$AG(e1 \Rightarrow AF_{\leq d} e2)$	The occurrence of $<exp1>$ always implies the occurrence of $<exp2>$ in less time than $<exp3>$.

Assertion	Formal description	Comments
reachable <exp1> [from <exp2>] [within <exp3>]	$AG(e1 \Rightarrow EF_{\leq d} e2)$	The occurrence of <exp1> may imply the occurrence of <exp2> in less time than <exp3>.
<exp1> after <exp2>	$AG(\neg EU(\neg e2, e1))$	<exp2> always occurs after <exp1>.

Using the syntax below, 3 equivalencies, with an increasing discrimination power, can be used :

- language equivalence : ensures that the observable executions of the abstract and real behaviors are the same. It takes into account neither the possibility of locking (states without any observable action) nor the possibility of divergence (infinite execution implying only non observable actions).
- observational equivalence : in addition to language equivalence, it takes into account locking states and the conservation of non determinism. Divergent executions are ignored.
- branching equivalence : in addition to observational equivalence, it takes into account divergent executions.

...

BEHAVIOR(<convention> equivalence)

STATES

<state name>(<state name>)* : **STATE**;

<state name> : **INITIAL STATE**;

TRANSITIONS

<transition 1>;

<transition 2>;

...

<transition n>;

END;

...

Let see now how the threads are modeled :

THREAD t

REQUIRES

sem1 : **DATA ACCESS** semaphore;

sem2 : **DATA ACCESS** semaphore;

```

END t;

-- definition of the thread family
THREAD IMPLEMENTATION t.t1
PROPERTIES
  Period => 13.96ms;
  cotre::Priority => 1;
  cotre::Phase => 0.0ms;
  Dispatch_Protocol => Periodic;

ANNEX cotre.behavior IS
STATES
  s0, s1, s2, s3, s4, s5, s6, s7, s8 : STATE;
  s0 : INITIAL STATE;
TRANSITIONS
  s0 -[ ]-> s1 { periodic_wait };
  s1 -[ ]-> s2 { COMPUTATION(1.9ms, 1.9ms) };
  s2 -[ sem1.wait ! (-1.0ms) ]-> s3;
  s3 -[ ]-> s4 { COMPUTATION(0.1ms, 0.1ms) };
  s4 -[ sem2.wait ! (-1.0ms) ]-> s5;
  s5 -[ ]-> s6 { COMPUTATION(2.5ms, 2.5ms) };
  s6 -[ sem2.release ! ]-> s7;
  s7 -[ ]-> s8 { COMPUTATION(1.5ms, 1.5ms) };
  s8 -[ sem1.release ! ]-> s0;
END ANNEX cotre.behavior;
END t.t1;

```

First, the thread type declares to need 2 semaphores in the core AADL syntax. The implementation t.t1 then shows an use of the COTRE properties `cotre::Priority` and `cotre::Phase`. The properties below, absent from the AADL core, are required for COTRE and defined explicitly in its property set :

Table 2. The COTRE property set

Name	Type	Applies to	Default value	Notes
cotre::Description	string	every component / sub-component	-	Informal comments.
cotre::Min_Time	time	THREAD	-	Minimum time between execution of a sporadic thread.

Name	Type	Applies to	Default value	Notes
cotre::Phase	time	THREAD	0.0s	Phase of the beginning of execution for periodic threads.
cotre::Priority	integer	THREAD	-	Base priority of the thread (the precise semantic depends on the scheduling policy).
cotre::Protected	boolean	SUBPROGRAM	false	true if the subprograms calls are exclusive, false if they are concurrent.
cotre::Reentrant	boolean	SUBPROGRAM	false	true if the subprogram is reentrant, and false if not.
cotre::Requirements	string	every component / sub-component		Requirements tracability informations.

The definition of the t.t2 implementation is identical excepted for the period property, which is 17.26ms.

Finally, here is the semaphore model :

DATA semaphore

PROVIDES

wait: **SUBPROGRAM**;

release: **SUBPROGRAM**;

PROPERTIES

cotre::Protected => TRUE; -- wait and release are exclusive

ENDsemaphore;

DATA IMPLEMENTATION semaphore.default

PROPERTIES

sem_p::Max_tokens => 1;

-- Ensures the number of tokens remains in the authorized range

```

ANNEX cotre.guarantees IS
  INVARIANT tokens < Max_tokens;
  INVARIANT tokens >= 0;
END ANNEX cotre.guarantees;

```

```

ANNEX cotre.behavior IS
  VARS
    tokens : INTEGER 0..+infinity;
  INITS
    tokens := sem_p::Max_tokens;

```

```

SUBPROGRAM wait

```

```

  STATES

```

```

    s0, s1 : STATE;
    s0 : INITIAL STATE;

```

```

  TRANSITIONS

```

```

    s0 -[ WHEN tokens > 0 => CALLED ? ]-> s1 { tokens := tokens -
1 };
    s1 -[ RESUME ]-> s0;

```

```

SUBPROGRAM release

```

```

  STATES

```

```

    s0, s1 : STATE;
    s0 : INITIAL STATE;

```

```

  TRANSITIONS

```

```

    s0 -[ WHEN tokens < capacity => CALLED ? ]-> s1 { tokens :=
tokens + 1 };
    s1 -[ RESUME ]-> s0;
  END ANNEX cotre.behavior;
END semaphore.default;

```

For any component, behavior is described thanks to Mealy machines. In the thread case, an automaton is globally linked to the object, when one is required for each provided subprogram for data components. Internal variables and their initial values have to be defined at the beginning of the annex. The states have then to be declared, and transitions are defined in this way :

```

(<label>:)* <origin state> -[<clearing condition>]-> <arrival state>
{<actions>;}

```

The clearing conditions are expressed as follow :

WHEN <boolean condition> => <synchronization event>

And the synchronization events known are :

Table 3. Synchronization events

Events	Comments
<subprogram name> ! [(<parameters>)]	Calls the named subprogram with the required parameters. The subprogram is identified by using the dotted notation <object> . <subprogram> . Parameters are separated by commas.
CALLED ?	The subprogram being described is called.
RESUME [(<parameters>)]	Give back the control to the caller (but the subprogram being described can go on running).

Finally, actions can be :

Table 4. Actions

Actions	Comments
COMPUTATION (<max_duration range>)	Consumes a CPU time smaller than <max_duration> or bounded by <range>.
DELAY (<max_duration or range >)	The execution is deferred for a time smaller than <max_duration> or bounded by <range>.
PERIODIC_WAIT	Delays the execution of a periodic thread until the beginning of its next period. For an aperiodic thread, it is equivalent to SKIP .
SKIP	Does nothing.
<l_exp> := <exp>	Modifies the value of the variable <l_exp>.

4. CONCLUSION

The COTRE project goal is to provide method and tools for real time software architecture design & verification. The approach adopted is based on formal languages and verification methods. The UCOTRE language is the user view of architectures, when VCOTRE is closer to verification

methods requirements. The proximity of the needs covered by AADL and UCOTRE allows to use the AADL core as a base language. The extension mechanisms defined in AADL are sufficiently powerful to take into account the COTRE specific concepts, such as behaviors and contracts. These COTRE developments are identified as possible annexes for future AADL versions. Currently, the UCOTRE language is quite clearly defined, even if some points are still under construction. But the semantic coherency is not completely established between UCOTRE and AADL on one hand, and between UCOTRE and VCOTRE on the other hand. These precise mappings are being studied, as well as between the HOOD graphical representation and UCOTRE. A demonstration toolkit, able to assist designers from the graphical modeling until the formal verification is also in progress.

5. REFERENCES

1. P. Farail, P. Gauffillet, J.M. Farines, J.L. Lambert, P. Dissaux, H. Hafidi, P. Michel, M. Filali, J.P. Bodeveix, F. Vernadat, B. Berthomieu, P.O. Ribet, The COTRE project : how to model and verify real time architecture ?, 2004
2. <http://www.laas.fr/COTRE>
3. <http://www.airbus.com>
4. <http://www.tni-valiosys.com>, <http://www.tni-world.com>
5. <http://www.feria.cnrs.fr>
6. <http://www.laas.fr>
7. <http://www.irit.fr>
8. <http://www.cert.fr/fr/dtim>
9. <http://www.enst-bretagne.fr>
10. http://forums.sae.org/access/dispatch.cgi/TEAASD_pf/folderFrame/100144/0/def/aac9
11. J.M. Farines, B. Berthomieu, J.P. Bodeveix, P. Farail, M. Filali, P. Gauffillet, H. Hafidi, J.L. Lambert, P. Michel, F. Vernadat, The COTRE project : rigorous software development for real time systems in avionics, 27th IFAC/IFIP/IEEE Workshop on Real Time Programming - Zielona Gora (Poland), 2003
12. Paul Peterson and Kim G. Larsen, Uppaal2k, European Association for Theoretical Computer Science, 70, 2000
13. K. L. McMillan, Symbolic Model Checking, Kluwer Academic, 1993

- 14.B. Berthomieu, P.O. Ribet, F. Vernadat, The tool TINA - Construction of abstract state spaces for Petri nets and time Petri nets, *Int. Journal of Production Research* (to appear)
- 15.S. Dellacherie, Samuel Devulder, Jean Luc Lambert, *Software Verification Based on Linear Programming*, World Congress on Formal Methods, 1999
- 16.<http://www.hood-method.org>
- 17.SAE AS-2C, Aerospace Information Report, Architecture Analysis and Design Language, version 0.96, 2004
- 18.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, J.L. Lambert, Towards the verification of real time systems in avionics : the COTRE approach, *Electronic notes in Theoretical Computer Sciences ENTCS* - Volume 80, 2003