Hierarchical Modeling and Verification of Timed Systems in Timed AltaRica

Claire Pagetti, Franck Cassez and Olivier Roux
IRCCyN - UMR CNRS 6597
1 rue de la Noë
BP 92101
44321 Nantes Cedex 3
France
{name@irccyn.ec-nantes.fr}

Abstract

In this paper we present a timed extension of the AltaRica language, Timed AltaRica, and describe the architecture of a compiler from Timed AltaRica to timed automata. We present the features of the language, namely modularity, hierarchical modeling and reuse of components during the specification phase, on an avionics example. Then, we use the compiler from Timed AltaRica to Timed Automata to check some safety properties on the system.

1 Introduction

Context. Nowadays computer programs are heavily used to automatically control *embedded systems* (e.g. network protocols, washing machines). Most of these systems are *critical* in the sense that a failure brings about losses of lives or of a huge amount of money. In the meantime these systems are becoming more and more complex and *high level languages* [16, 8] have been designed to specify those systems. Those languages usually provide support for *hierarchical* (or modular) specification. Once specified there is a real need to check that the system meets some *requirements*. In the last two decades major breakthroughs have been achieved in the area of *formal verification*: i) it has become usual to verify (parts of) industrial discrete systems via model-checking [21] (ii) more recently the verification techniques [19] have been extended to *continuous systems* e.g *timed* and *hybrid automata* [24, 17]. Nevertheless if is often the case that specification languages provide no support for real-time (continuous) specification (e.g. SDL [14]) or that no proper formal verification can be carried out on the specification (e.g. UML [11]). On the other hand, the timed models like timed and hybrid automata [18] are difficult to use as specification languages as they barely support *hierarchical* specification.

A Hierarchical Specification Language: AltaRica. Modern software design exploits *hierarchy* as a major feature for specifying complex systems. Usually a hierarchical specification language provides a construct to define (elementary) *components* that can be reused and *composed* (or synchronized) to create more complex components. The expressiveness of such languages is determined by the power of the synchronization mechanism. In the 80's a lot of languages were developed to handle hierarchy (see [16]). AltaRica [15, 7] emerged at the same time and is now a hierarchical specification language based on *constraints automata* [12] and is equipped with a powerful synchronization mechanism for interacting components, including priorities for instance. It has been used in various flavors to develop industrial software for instance by the french plane manufacturer Dassault Avionics, the french power supply company Electricit de France or the oil company TotalElfFina. A toolkit is currently being developed (see Figure 1) at the University of Bordeaux I, France, in the AltaRica project, with the following features:

The AltaRica specification language: The syntax and semantics of the language are formally defined [7]; the basic objects of the language are *components* (finite state constraints automata) and can be hierarchically composed to create *nodes*. The composition mechanism is quite powerful and extends the well-known *synchronized product* (la Arnold-Nivat) [1]. From this specification language one can extract different types of information.

The AltaRica compilers: An AltaRica program can be compiled into a *finite state automaton* [6, 26] on which formal verification can be carried out; another compiler can produce a *fault-tree* on which reliability analysis can also be carried out [5, 27]; a third compiler produces a *stochastic Petri Net* on which performance analysis can be performed [28].

AltaRica is thus a powerful hierarchical specification language with formal verification support. The need for *real-time specification* and *verification* emerged from the industrial users of AltaRica.

Our Contribution. We have been involved in the AltaRica project for two years and our work consists in *extending* AltaRica with *real-time*. This work is two-fold: i) first we had to extend the syntax and semantics of AltaRica with real-time features [13, 22]; the extended language is called Timed AltaRica; ii) second we had to provide a way of *verifying* real-time AltaRica specifications. We have implemented a tool TARC that translates a Timed AltaRica specification into a timed (or hybrid if needed) automaton that extends the AltaRica Toolkit (Figure 1).

In this paper we describe the use of Timed AltaRica to specify real-time systems: we take an avionics example from [10, 9]. Then we show how to use our compiler to produce a timed automaton from a Timed AltaRica program; finally we verify safety properties using the tool UPPAAL [24] for analyzing timed automata.

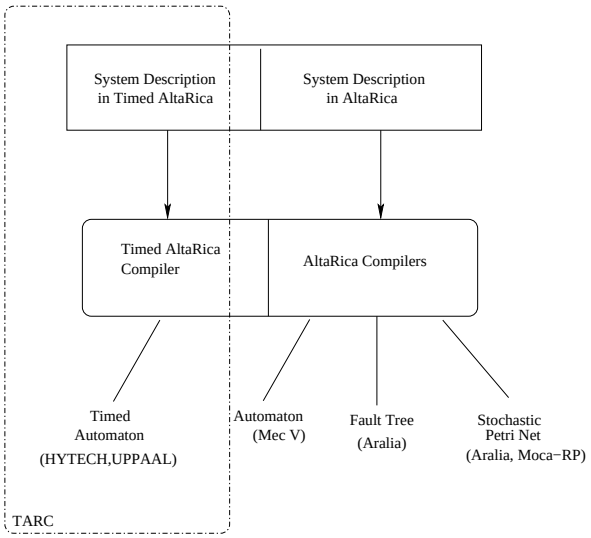

Figure 1: Overview of the AltaRica Toolkit

Related work. UML [11] provides no support nor for real formal verification neither for real-time specifications. SDL [14] has no support for real-time specification. Argos [20] is an extension of *StateCharts* [16] but the timing features of Argos are restricted: one can only constrains the time the system stays in a state (the *invariants* of the timed automata model) and consequently the expressive power of Argos is strictly less than timed automata. On the contrary Timed AltaRica has is as expressive as timed automata. Mocha [3] is another specification language but to our knowledge does not provide with real-time specification so far (although it is planned to add it to the language). Charon [4] is quite similar to AltaRica and provides with a hierarchical specification language for hybrid systems. The time features (the dynamics the specification language handle) of Charon are more advanced than in Timed AltaRica but there is no mechanism for easily specifying *timed priorities* among components.

Outline of the Paper. In section 2 we describe the *steering control system* of an airplane. We give its Timed AltaRica specification and show how the constructs and features of the language allows for a clear and easy way to specify such a system. In section 3 we describe the architecture of the Timed AltaRica compiler that translates Timed AltaRica specification into timed (and hybrid) automata. We particularly focus on two steps: how to cope with hierarchy and how to translate a simple component into a timed automaton. We finally show how to use the tool UPPAAL [24] to check some properties of the designed steering control system. In section 5 we discuss some ongoing and futur work.

2 Specification of the Steering Control with Timed AltaRica

2.1 Informal Specification

We present an example from the avionics area developed in [9] and partially treated in [10] to illustrate some Timed AltaRica features. We consider the part of the system in charge of controlling the right steering of the plane (see Figure 2) and take the informal specification of [10]:

- Three computers F_R, F_L and F_B are in charge of controlling the steering;
- Two main buses are used: C_{LR} (bold line on Figure 2) connects F_R and F_L; C_{LRB} (dashed line on Figure 2) connects F_B to F_R and F_B to F_L. There is another bus from F_B, F_R, F_L to the steering but we do not consider it and assume the communication on this bus is instantaneous.

The informal specifications are as follows:

1. The steering is controlled by the *Cmd* event. *Cmd* can be issued either by F_R or F_L or F_B and is sent through the instantaneous bus.
2. Initially F_R controls the steering and F_L and F_B are idle (they do not issue any *Cmd*).
3. If F_R fails then F_L takes over the control. If F_L fails, F_B takes over the control. In case F_L has already failed when F_R fails the control is handled directly by F_B. The assumption is that F_B cannot be down.
4. Each computer does a sampling control of the steering: when in charge of the control, they issue a *Cmd* event every 20 ms.
5. Every 20 ms C_{LR} updates the value of a state variable that corresponds to the state of F_R (0 for non failure and 1 for failure).
6. The bus C_{LRB} works similarly and updates every 40 ms the values of a state variable that corresponds to the number of components F_R and F_L that have failed (either 0, 1 or 2).
7. The recovery scheme is the following: F_L scans the last updated value of C_{LR} every 20 ms; when it is 1 it takes over. There is a delay between 10 and 20 ms before the take over is active. F_B scans the last updated value of C_{LRB} every 20 ms and when the sum is 2 it takes over[1].

The requirements for the control of the steering are given by the following properties:

[1] As already mentioned F_B cannot be down so this scheme is comprehensive.

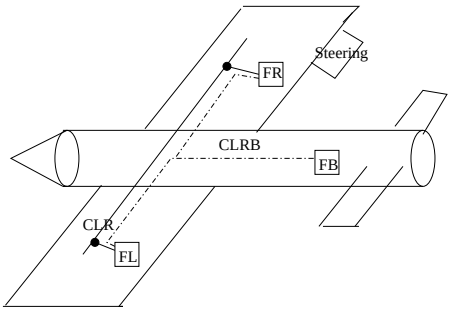

Figure 2: Architecture of the Steering Control Network

P_1: at most one computer at a time can send the *Cmd* event;

P_2: the maximum delay between two *Cmd* events must be less than 160 ms.

We now show how to model this control system with Timed AltaRica and how to check the requirements.

2.2 Timed AltaRica Specification

Specification of the buses. The basic component in Timed AltaRica is the *node*. A node consists of 3 main parts: the first one is the *declaration* of the variables (types, range) (see node `CLRB` Figure 3, lines 2–8). Then comes the *transitions* part (lines 9–10) together with the initial state (lines 11–12). The last one is the *assertion* part that constrains the values of the variables (lines 13–16).

In the example of the node `CLRB` (Figure 3) we use one discrete state variable (line 5), `failure` that gives the value of the last update defined by the transition of line 10. Such variables are *local variables* and if one wants to make their values available from the outside of the node, a *flow* variable can be used (lines 2–3). For instance `Failure_RLB` is a flow variable and it is constrained (line 14) in node `CLRB` to track the value of the variable `failure`. `Failure_L` and `Failure_R` are (external) flow variables and are shared together with `Failure_RLB` with the other components. We just point out that if an assertion of the form `Failure_R=0` is used in one component and `Failure_R=1` is used in another then the system will have no states at all as it is impossible to meet this specification. So far it is the user's task to check that flow variables are not incorrectly used. One way to ensure this is to force that the assignment of a flow variable occurs only in one component.

In the Timed AltaRica language we can use *clock* variables (and clock flows) like `h` (line 6). It is exactly the same as the clocks used in timed automata in UPPAAL [24]. One can reset a clock variable: this happens on the transition of line 10. To fire this transition we have to wait until `h=40` this means 40 time units since the last time `h` was reset. The notion of *invariants* from timed automata is implemented via the `tinvariant` assertions (lines 15–16): time can

progress as long as the invariants are true, which means in case of node `CLRB` that at most 40 time units can elapse between two `refresh` transitions. The node `CLRB` implements the specification item (6) of Section 2.1: it reads every 40 time units the values of `Failure_L` and `Failure_R` (that tells if a node F_R or F_L has failed) and sets `failure` accordingly. Notice that the assertion `Failure_RLB=failure` makes available the updated value to the other components. The specification of the bus C_{LR} (item (5) of Section 2.1) is quite similar and is given in appendix A, Figure 8, page 78.

```
1   node CLRB
2     flow
3        Failure_L , Failure_R : [0,1]; Failure_RLB : [0,2]
4     state
5         failure : [0,2];
6         h : clock
7     event
8         refresh
9      trans
10         h=40 |- refresh -> h:=0,  failure:= Failure_L + Failure_R
11    extern initial_state =
12            failure=0 & h=0
13    assert
14           Failure_RLB=failure
15    tinvariant
16          true => (h<=40);
17    edon
```

Figure 3: Timed AltaRica Specification of the C_{LRB}

Specification of the Computers. The specification of the computer F_R (items (1) and (4) of Section 2.1) is straightforward and given in appendix A, Figure 9, page 79. A flow variable $Failure_R$ informs the others if the component F_R is broken down (in which case $Failure_R = 1$). As long as $Failure_R = 0$, the component sends a Cmd event each 20 t.u. which is specified by the timed invariant `(loc=0 => x <=20)`.

The specifications of the computers F_L and F_B are a little bit more involved and are respectively given in the Figure 6 page 73 and in the appendix A, Figure 10 page 79. As for the component F_R, F_L communicates its state through a flow variable $Failure_L$. Moreover, we assume that F_B cannot break. For F_L and F_B, an internal variable $mode$ gives the mode of the component: $mode = 0$ corresponds to an idle state, that means it does not send any Cmd event; if $mode = 1$ the component starts the computation for the command and for $mode = 2$ the component controls the steering and sends the Cmd event each 20 t.u.

Putting all Together. The Timed AltaRica specification of the steering is given in appendix A, Figure 11, page 79. Every time the steering receives a Cmd event the local clock z is reset. With this specification, the property P_2 (Section 2.1) reduces to checking that the clock z of this node is always less than 160.

Now we can define the system in a hierarchical way and build a node `main` that synchronizes

the (sub)nodes we have defined previously. The subcomponents are first instantiated (Figure 4, lines 2–8). This mechanism allows the programmer to reuse previously defined components (e.g. two steerings could be defined by `st1, st2 : STEERING`). Next we define the synchronization of the subcomponents (item (1) of the informal specs, Section 2.1) and here we use a powerful feature of AltaRica: *broadcasting*. The synchronization macro-vector of line 10 defines a set of synchronization vectors and some priorities among them:

1. The constraint ≥ 2 imposes that at least two events among the four must occur simultaneously. For instance[2] `<st.cmd,-,fl.cmd,->` is suitable as well as `<st.cmd,-,fl.cmd,fb.cmd>`. The macro-vector `<st.cmd,fr.cmd?,fl.cmd?,fb.cmd?> >=2` generates 7 synchronization vectors.

2. In case two transitions with synchronization vectors v and v' are simultaneously enabled, the priority is given to the one that involves the largest number of nodes (if v =`<st.cmd,-,fl.cmd,->` and v'=`<st.cmd ,-,fl.cmd,fb.cmd>` are both fireable in a state of the system, only v' will be fired).

The last part of node `main` contains some *flow coordination* assertions (lines 12-16): it constrains the values of the flow variables of the sub-nodes.

```
1   node main
2     sub
3       fr : FR;
4       fl : FL;
5       cr : CLR;
6       cs : CLRB;
7       fb : FB;
8       st : STEERING
9     sync
10     <st.cmd,fr.cmd?,fl.cmd?,fb.cmd?> >=2;
11    assert
12      fr.Failure_R=cr.Failure_R;
13      fl.Failure_LR=cr.Failure_LR;
14      fr.Failure_R=cs.Failure_R;
15      fl.Failure_L=cs.Failure_L;
16      fb.Failure_RLB=cs.Failure_RLB
17    edon
```

Figure 4: Timed AltaRica Specification of the Whole System

In this example we have not used any *timed priority*. Nevertheless the Timed AltaRica language allows the user to define easily many types of time priorities like *urgency*. The reader is referred to [13, 22] for a detailed description.

[2]In the sequel, the notation `node.var` refers to the variable `var` in the node `node`.

3 The Timed AltaRica Compiler

An AltaRica program is hierarchical and built from nested sub-nodes. A useful theorem ([7]) shows how to rewrite a hierarchy of nodes into a single (flat) node while preserving the semantics of the hierarchical specification. This process is called *flattening* and is the key operation of the AltaRica compilers. MEC V [29] is one of those compilers and translates efficiently a hierarchical (untimed) AltaRica program into a transition relation encoded with Binary Decision Diagrams (BDD).

We have extended the previous flattening theorem for Timed AltaRica programs [13]. The flattening operation is quite involved and our compiler TARC (Timed AltaRica Compiler) for Timed AltaRica is built upon MEC V [29] in order to reuse the efficient flattening procedure.

3.1 Architecture of the TARC Compiler

We now describe how TARC works:

1. **Abstraction** of the timed constraints: in a first step TARC substitutes a boolean variable for every elementary timed formula. In the example of Figure 3, we replace $h = 40$ by γ_1, $h \leq 40$ by γ_2 and we need another variable γ_3 for the assignment `h:=0` (just think `h:=0` is encoded as `h'=0` where h' gives the value of h after the transition has been fired.) The initial condition does not generate any new variable as this is also γ_3. This gives an **untimed** node `FL_u`. In the case of a hierarchical description we do this abstraction for each node.

2. **Flattening**: here we use the compiler MEC V to do the flattening on the untimed nodes obtained by abstraction (step 1). The result of the compilation is a set of triples (B_1, e, B_2) where $B_i, i \in [1, 2]$ are BDDs encoding sets of values of discrete variables of the nodes, and e is an event of the node `main`. For each $v \in B_1$ there exists $v' \in B_2$ such that $v \xrightarrow{e} v'$ is a transition of the node `main`.

3. **Concretization**: from the previous untimed transition relation, we want to compute a timed automaton. This is not straightforward and the idea is to partition the discrete state space into classes that satisfy the same time invariants. This process is formally described in [13]. It leads to the smallest (in terms of numbers of locations) timed automaton that is timed bisimilar to the initial Timed AltaRica node. The timed automaton obtained from the node F_L (appendix A, Figure 6, page 73) is given in Figure 7.

The TARC compiler produces UPPAAL [24] or HyTech [17] input files from a Timed AltaRica program. The abstraction step is quite clear and in the sequel, we detail the flattening and concretization phases.

3.2 Flattening

AltaRica and Timed AltaRica are two hierarchical and modular languages so that any AltaRica and Timed AltaRica specification can be reduced into simpler specification without any modeling features such as priority or broadcast. This result has been proved in a *rewritting theorem* [25, 13]: any AltaRica (resp. Timed AltaRica) node with subnodes and synchronisation (even broadcast) constraints can be rewritten syntactically into a bisimilar AltaRica component (resp. timed bisimilar Timed AltaRica component) without subcomponent and priority (resp. timed priority). Then any Timed AltaRica program can be flattened down into a Timed AltaRica component which is semantically equivalent and behaves in the same way. This allows to apply formal methods on the system, such as model checking, and to verify exhaustively some properties.

Practically, a Timed AltaRica program is abstracted (phase 1) and the MEC V Compiler [29] flattens the abstracted node. Any component is encoded into a BDD, more precisely the BDD encodes the transition relation of the component. Any node is translated inductively: the BDD associated to a node with subcomponents is the union of the components' BDDs and the node BDD (representing its local behavior) restricted by all the constraints such as the synchronised vectors and assertions.

In our example, the flattened automaton has exactly 4 locations. We use the TARC compiler on the specification of the steering control and we obtain the flattened timed automaton (it has 25 transitions) given in the Figure 5.

3.3 Concretization

Once the flattening is achieved, we need to put back some abstracted information related to quantitative timing constraints. We illustrate the translation from Timed AltaRica to Timed Automata with the component F_L given in the Figure 6. The formal algorithm is given and proved in [13]. For the node F_L we obtain the UPPAAL timed automaton given in the Figure 7.

Timed Automata. First, let us recall some notions on timed automata [2]. A timed automaton is a tuple $(Q, E, q_0, X, I, \rightarrow)$, where Q is set of *locations*, E is the set of *actions*, q_0 is the *initial location*, X is the set of *real-time clocks*, I is a mapping $I : Q \rightarrow \mathcal{C}(X)$ where $\mathcal{C}(X)$ is a set of constraints on the clocks, that associates to each location a formula on clocks called a *timed invariant* and $\rightarrow \subseteq Q \times \mathcal{C}(X) \times (E \cup \mathbb{R}_{\geq 0}) \times 2^X \times Q$ is the *transition relation*. Figure 7 depicts a timed automaton with two locations $Q = \{s_0, s_1\}$ and one clock $X = \{h\}$. The behavior of a timed automaton consists of (i) continuous time step: time progresses and the clocks as well and (ii) discrete transitions from one location to another taking no time.

For the timed automaton of Figure 7 the initial location is s_1. There is no invariant on s_0, so that $I(s_0) = true$. It means that the automaton can stay in the location s_0 forever. On the other

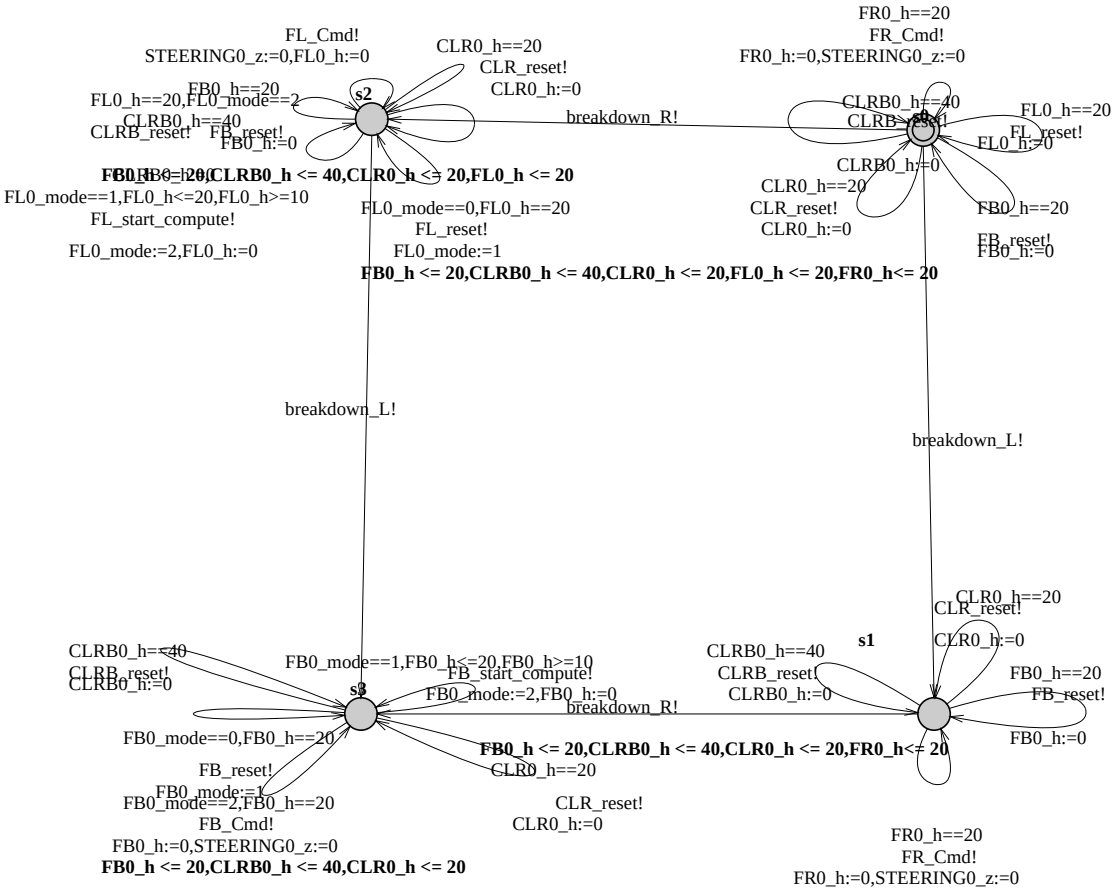

Figure 5: UPPAAL Flattened Timed Automaton of the System

```
node FL
  flow
     Failure_L , Failure_LR : [0,1];
  state
      failure : [0,1];
      mode : [0,2] ;
      h : clock
  event
      reset, cmd, start_compute, breakdown_L
   trans
       mode=0 & Failure_LR=0 & h=20 & failure=0 |- reset -> h:=0 ;
       mode=0 & Failure_LR=1 & h=20 & failure=0 |- start_compute -> h:=0, mode:= 1;
       mode=1 & h<=20 & h>=10 & failure=0 |- cmd -> h := 0, mode:=2 ;
       mode=2 & h=20  &  failure=0 |- cmd -> h := 0;
       true |- breakdown_L -> failure := 1;
  extern initial_state =
          mode=0 & failure=0 & h=0
  assert
         Failure_L=failure
  tinvariant
        (failure=0) => (h<=20);
 edon
```

Figure 6: Timed AltaRica Specification of the computer F_L

hand, the formula $h \leq 20$ is the invariant of s_1, i.e. $I(s_1) = h \leq 20$, and then the system can stay in s_1 only as long as $h \leq 20$ (e.g. starting with $h = 10$, time can elapse at most for 10 t.u. in s_1). There are 4 events, one on each transition. For instance the transition $(s_1, h = 20, reset, h, s_1)$ is a loop on s_1 that can be fired if the *guard* $h = 20$ is true; when the transition is fired h is reset. Furthermore, we consider timed automata with discrete variables so that we can extend the guards and the assignment on the variables (e.g. $k := k + 1$). These timed automata with variables have the same expressiveness as timed automata as long as their domain is bounded and can be analysed by the model-checking tools like UPPAAL and HYTECH.

From Timed AltaRica to Timed Automata A state of a Timed AltaRica component is a valuation of its variables and obviously, a rough manner to transform a Timed AltaRica program into a timed automaton consists in associating a location to each valuation of the set of discrete variables, and then to add the timed invariant accordingly. This procedure may produce very large timed automata.

Our strategy is to group together some states and produce the least possible number of locations. We briefly describe the algorithm that implements this (the formal algorithm is detailed in [13]). First the timed invariant of the component F_L Figure 6 `(failure=0 => h<=20)` partitions the set of valuations into two parts: either $failure = 0$ or not. In the first case, all the valuations such that $failure = 0$ must verify the timed invariant $h \leq 20$ whereas the valuations such that $failure \neq 0$ has no timed constraint.

From this we create two locations s_0 and s_1 with invariants $I(s_0) = true$ and $I(s_1) = h \leq 20$. Notice that we need some discrete variables in our timed automata: $\{mode, failure, Failure\_R, Failure\_LR\}$.

Now we try to compute the transition relation from the set of transitions of the Timed AltaRica specification. Consider the transition:

```
mode=0 & Failure_LR=0 & h=20 & failure=0 |-  reset ->   h:=0
```

This transition does not modify any discrete variables: if we start in location s_i we end up again in s_i. Note that $failure = 0$ implies this transition is only fireable from s_1. It is depicted at the bottom of s_1 on Figure 7. (notice that this is UPPAAL automaton and that the events that are not broadcasted are abstracted away, which is the case for event *reset*).

Now consider the transition

```
mode=0 & Failure_{LR}=1 & h=20 & failure=0 |- start_compute ->
                      h:=0, mode:= 1
```

Again this transition is not fireable from s_0 because of the constraint $failure = 0$. The only outgoing transition is from s_1 and is depicted at the top of state s_1.

The same method leads to the transition labeled *cmd*! (send event Cmd) from s_1 to s_1.

Note that we may have to iterate and refine the partition we have started with (s_0 and s_1): the detailed procedure is given in [13]. We finally obtain the automaton of Figure 7.

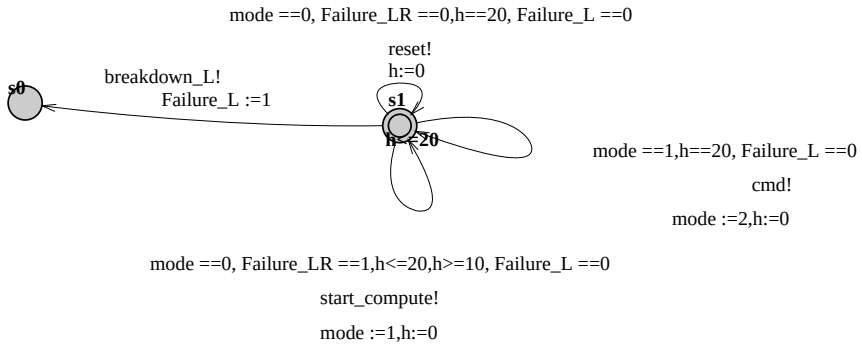

Figure 7: UPPAAL Timed Automaton of F_L

4 Verification

We can know verify the requirements given in the section 2.1 page 66. We check the following properties with UPPAAL:

- the safety property `A[](not deadlock)` is true which means our timed automaton has no deadlock;
- we can encode P_1 in UPPAAL input temporal logic, as `A[](FR.s1 imply (FL.mode==0 and FB.mode==0) and ((FL.mode==2 and FL.s 1) imply FB.mode==0))`. It is also satisfied by the system;
- the UPPAAL temporal logic encoding of P_2 (Section 2.1) is `A[](steer- ing.z <= 160)` and again is true.

Moreover we can check with UPPAAL that the upper bound 160 ms is actually reached and UPPAAL produces a witness of this behavior.

5 Conclusion

We have implemented a translation from Timed AltaRica to timed automata and hybrid automata. The compiler TARC is based on the formal semantics of Timed AltaRica given in [13]. A prototype version of the compiler [23] is available and currently being improved.

With this prototype compiler we have demonstrated the usefulness and power of the Timed AltaRica language. The Timed AltaRica compiler produces either UPPAAL [24] or HyTech [17] files which enables us to use these efficient tools to check timing requirements. This work is a significant contribution to the AltaRica Workbench and will enable the users to specify more accurately their models and to check for real-time properties.

Acknowledgements The authors wish to thank Aymeric Vincent, LaBRI Bordeaux, France, for his help in designing the TARC Compiler, the source code of which is based on [29].

References

[1] Arnold A. *Finite transition systems.* Prentice-Hall. Prentice-Hall, 1994.

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

[3] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *CAV 98: Computer-Aided Verification*, Lecture Notes in Computer Science 1427, pages 521–525. Springer-Verlag, 1998.

[4] Rajeev Alur, Radu Grosu, Yerang Hur, Vijay Kumar, and Insup Lee. Modular specification of hybrid systems in CHARON. In *HSCC*, pages 6–19, 2000.

[5] Groupe ARALIA. Computation of prime implicants of a fault tree within aralia. *Reliability Engineering and System Safety*, 1996. Special issue on selected papers from ESREL'95.

[6] A. Arnold, D. Begay, and P. Crubille. *Construction and analysis of transition systems with MEC*. World Scientific, 1994.

[7] A. Arnold, A. Griffault, G. Point, and A. Rauzy. The altarica formalism for describing concurrent systems. *Fundamenta Informaticae*, 40:109–124, 2000.

[8] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, september 1991.

[9] F. Boniol, G. Bel, and J. Ermont. Modélisation et vérification de systèmes intégrés asynchrones : étude de cas et approche comparative. *9ème Conférence Internationale sur les Systèmes Temps Réel, RTS'2001*, 2001.

[10] F. Boniol and J. Ermont. Modelling and verifying embedded systems sensitive to resource availability. *3rd European Systems Engineering Conference, EuSEC'2002*, 2002.

[11] Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[12] S. Brlek and A. Rauzy. Synchronization of constrained transition systems. In *ed. H. Hong, Proc. of the First International Symposium on Parallel Symbolic Computation – PASCO'94*, World Scientific Publishing, pages 54–62, 1994.

[13] F. Cassez, C. Pagetti, and O. Roux. A timed extension for altarica. Technical Report R 12002-13, IRCCyN/CNRS, Nantes, 2002.

[14] CCITT. *CCITT Recommendation Z.100: Specification and Description Language SDL,Blue Book*, volume X.1-X.5. ITU General Secretariat, Geneva, 1988.

[15] A. Griffault, S. Lajeunesse, G. Point, A. Rauzy, J.-P. Signoret, and P. Thomas. The altarica language. In *Proceedings of the International Conference on Safety and Reliability, ESREL'98*. Balkema Publishers, June 20-24 1998.

[16] David Harel. STATECHARTS- *a Visual Formalism for Complex Systems*, volume 8, pages 231–274. North Holland, june 1987.

[17] T. A. Henzinger, P. H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *Lecture Notes in Computer Science*, 1254:460–473, 1997.

[18] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57(1):94–124, August 1998.

[19] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, England, 2000.

[20] M. Jourdan and F. Maraninchi. Static timing analysis of real-time systems. In *ACM Sigplan Workshop on Languages, compilers and tools for real-time systems*, number 11 in ACM SIGPLAN Notices, San Diego, CAL, November 1995. ACM.

[21] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academics Publishers, 1993.

[22] C. Pagetti. Une extension temporise d'AltaRica. In *Actes du Congrs Modlisation des Systmes Ractifs, MSR'03*, 2003. forthcoming.

[23] Claire Pagetti. The Timed Altarica compiler, 2003. available at `http://altarica.labri.fr`.

[24] P. Pettersson and K. Larsen. UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, February 2000.

[25] G. Point. *Altarica : Contribution à l'unification des méthodes formelles et de la sûreté de fonctionnement*. PhD thesis, University of Bordeaux I, Janvier 2000.

[26] A. Rauzy. Toupie = mu-calculus + constraints. In *Proceedings of Computer Aided Design, CAV'95*, volume vol.939 of *Lecture Notes in Comput. Sci.*, pages 114–126, 1995.

[27] A. Rauzy. Mode automata and their compilation into fault trees. *Reliability Engineering and System Safety*, 78:1–12, 2002. URL: http://iml.univ-mrs.fr/ arauzy/publis/publis.html.

[28] J.-P. Signoret. Moca-rp v9. Technical Report EP/P/SE/MRT-ARF/JPS9634, Elf-Aquitaine, 1995. simulation de Monte-Carlo de rseaux de Petri stochastiques.

[29] Aymeric Vincent. The MEC V compiler, 2003. available at `http://altarica.labri.fr`.

A Timed AltaRica Specifications

```
node CLR
  flow
     Failure_R , Failure_LR : [0,1];
  state
      failure : [0,2];
      h : clock
  event
      refresh
   trans
       h=20 |- refresh -> h:=0, failure:= Failure_R ;
  extern initial_state =
          failure=0 & h=0
  assert
         Failure_LR=failure
  tinvariant
         true => (h<=20);
 edon
```

Figure 8: Timed AltaRica Specification of the C_{LR}

Remark. The flow variable `Failure_L` indicates whether F_L has failed or not. The node reads the value of the flow variable `Failure_LR` updated by the bus C_{LR} (node `Failure_CLR`, Figure 8.) The node `FL` can breakdown anytime as specified by the transition of line 15. In the initial state `mode=0` and F_L is idle (does not send any `cmd` event). It reads the value of `Failure_LR` every 20 time units as imposed by lines 11–12 and 21. If it has not failed and reads that F_R has failed (line 12) it switches to mode 1. In this mode it takes some time (between 10 and 20 time units) before it is actually ready to take over and send the event `cmd`. It then sends it every 20 time units (unless if fails). Notice that the time invariant `h<=20` only applies when `failure=0`: once the computer has failed no time invariant applies; otherwise time would be prevented from elapsing when `failure=1` and the whole system would deadlock (fortunately such deadlocks can be checked for with UPPAAL [24]).

The specification of the node F_B is quite similar and is given in Figure 10.

```
node FR
  flow
     Failure_R : [0,1]
  state
     loc : [0,1];
     x : clock
  event
     breakdown_R, cmd
  trans
     x=20 & loc=0 |- cmd -> x:=0;
     x<=20 & loc=0 |- breakdown_R -> loc := 1
  assert
        Failure_R=loc
  tinvariant
       (loc=0) => (x <=20)
  extern initial_state =
         loc=0 & x=0
 edon
```

Figure 9: Timed AltaRica Specification of the computer F_R

```
node FB
  flow
     Failure_RLB : [0,2];
  state
      mode : [0,2] ;
      h : clock
  event
      reset, cmd, start_compute
   trans
       mode=0 & Failure_RLB<2 & h=20 |- reset -> h:=0 ;
       mode=0 & Failure_RLB=2 & h=20 |- start_compute -> h:=0, mode:= 1;
       mode=1 & h<=20 & h>=10 |- cmd -> h := 0, mode:=2 ;
       mode=2 & h=20  |- cmd -> h := 0;
  extern initial_state =
          mode=0 &  h=0
  tinvariant
        (mode = 2 | mode =1 | mode= 0) => (h<=20)
  edon
```

Figure 10: Timed AltaRica Specification of the computer F_B

```
node STEERING
 state loc : [0,1]; z : clock
 event cmd
 trans loc=0 |- cmd -> z:=0
 extern initial_state =
     loc=0,z=0
edon
```

Figure 11: Timed AltaRica Specification of the Steering