Automated generation of minimal cut sets from AltaRica 3.0 models

Tatiana Prosvirnova*

LIX – Ecole Polytechnique – Route de Saclay,
91128 Palaiseau cedex, France
Email: prosvirnova@lix.polytechnique.fr
*Corresponding author

Antoine Rauzy

Chaire Blériot-Fabre – Ecole Centrale de Paris – Grande voie des, vignes, 92295 Châtenay-Malabry cedex, France
Email: Antoine.Rauzy@ecp.fr

Abstract: In this article, we present an algorithm to generate minimal cut sets from AltaRica 3.0 models. AltaRica 3.0 improves the previous versions of the language by introducing a fixpoint mechanism to stabilise values of variables after each transition firing. This fixpoint mechanism allows the design of acausal models and the analysis of systems with instant loops. It makes however the generation of fault trees more complex. We show here that by using advanced partitioning techniques, we can nevertheless design an efficient generation algorithm. We illustrate the different steps of this algorithm by means of a red wire example.

Keywords: automated generation of Fault Trees; minimal cut sets; model-based safety analysis; AltaRica.

Reference to this paper should be made as follows: Prosvirnova, T. and Rauzy, A. (2015) 'Automated generation of minimal cut sets from AltaRica 3.0 models', *Int. J. Critical Computer-Based Systems*, Vol. 6, No. 1, pp.50–80.

Biographical notes: Tatiana Prosvirnova is a PhD student in the Computer Science Laboratory of Ecole Polytechnique (LIX), Palaiseau, France. Her research is focused on the model-based approach for safety analyses. She is involved in the specification of AltaRica 3.0 modelling language and works on the development of the assessment tools for guarded transition systems, the underlying formalism of AltaRica 3.0. Graduated from Ecole Polytechnique in 2008, she has been working for three years as a Research Engineer in Dassault Systems. She worked on the development of the workshops to perform safety analyses (safety designer and Aralia fault tree analyser).

Antoine Rauzy is a Full Professor at Ecole Centrale de Paris and the Head of the Chair Blériot-Fabre sponsored by SAFRAN group. He is also an Associate Professor at Ecole Polytechnique. During his career, he moved back and forth from the academy to industry. His main scientific contributions stand in safety and reliability engineering, modelling language and assessment algorithms. He has published over 100 articles in international conferences and journals on these topics. He also designed state-of-the-art tools for fault tree analyses as

well as the AltaRica modelling language. He has organised many scientific conferences and is regularly on programme committees and editorial boards of reference journals and conferences of his domain.

1 Introduction

Fault Trees are probably the most popular formalism to support probabilistic risk and safety analyses. Efficient algorithms have been designed to assess these models (see, e.g., Rauzy, 2008a, 2012). Mature commercial tools are available. Fault Trees are however quite far from system specifications. They incorporate implicitly a large amount of knowledge of the analyst. As a consequence, they are hard to maintain throughout the life cycle of systems.

In order to tackle this problem, several authors proposed to generate Fault Trees from higher level descriptions (see, e.g., Majdara and Wakabayashi, 2009; for a recent review). In this way, not only safety models are closer to system specifications, but also the same high level model can be used to study several failure conditions and/or plant configurations. Most of the proposed approaches (including the one of the above reference) rely on various extensions of reliability block diagrams. Basic blocks of the diagram carry out both failures and inputs/outputs relations. The flow circulating in the diagram is analysed backward to generate the Fault Tree. This idea stems from artificial intelligence tools such as assumption-based truth maintenance systems (de Kleer, 1986).

The high level modelling language AltaRica (Point and Rauzy, 1999; Arnold et al., 2000; Boiteau et al., 2006) generalises also reliability block diagrams. It is however essentially a formalism to describe state machines and to compose them. It is therefore much more expressive than all extensions of reliability block diagrams proposed so far. Several integrated modelling and simulation environments have been developed to support the authoring and the analysis of AltaRica models. A versatile set of processing tools has been designed such as compilers to Fault Trees (Rauzy, 2002), compilers to Markov chains (Brameret et al., 2013), model-checkers (Griffault and Vincent, 2004) or stochastic simulators (Khuu, 2008). Quite a few successful applications have been reported in the literature (see, e.g., Bernard et al., 2007; Sagaspe and Bieber, 2007; Bieber et al., 2008; Bernard et al., 2008; Humbert et al., 2008; Quayzin and Arbaretier, 2009; Sghairi et al., 2009; Chaudemar et al., 2009; Adeline et al., 2010).

AltaRica 3.0 is a new version of the language (Prosvirnova et al., 2013). Its new underlying mathematical model, Guarded Transition Systems (GTS) (Rauzy, 2008b; Prosvirnova and Rauzy, 2012), improves the expressive power of the previous versions by introducing a fixpoint mechanism to stabilise values of flow variables after each transition firing. This mechanism allows the design of acausal components and the treatment of systems with instant loops. Because of its expressiveness, AltaRica 3.0 cannot be compiled into Fault Trees just by backward induction on values of flow variables. More elaborated compilation schemes must be applied. We show here that it is possible to keep this generation efficient thanks to advanced partitioning techniques.

The main contribution of this article stands in the step by step description of the compilation algorithm by applying it on a red wire example. The ideas developed here can be applied to other modelling formalisms, beyond AltaRica and GTS.

The remainder of this article is organised as follows. Section 2 presents the red wire example, a simple network system, containing loops and bidirectional flows. Section 3 gives an overview of the AltaRica 3.0. Section 4 presents the compilation algorithm. Section 5 gives some numerical results. Section 6 discusses related works. Finally, Section 7 concludes the article and outlines directions for future works.

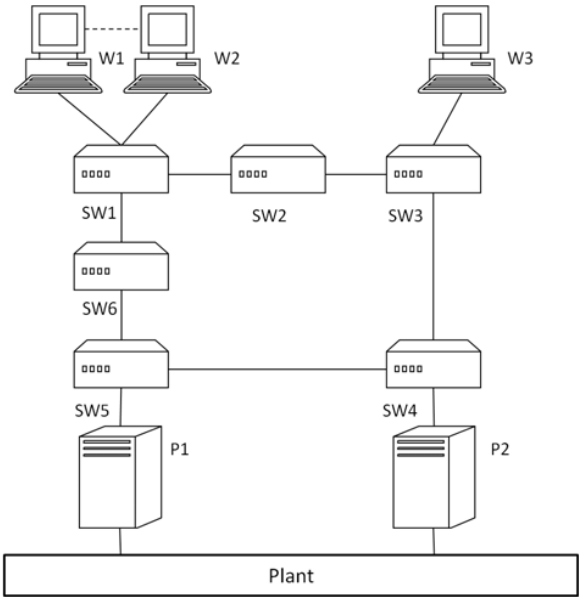
2 Motivating example

Consider the network depicted in Figure 1, which is inspired from Riera et al. (2012). This network is made of:

- three workstations W1, W2 and W3 producing data
- two processing units P1 and P2 in charge of processing data
- six switches SW1, …, SW6 receiving data from workstations and/or other switches and transmitting them to processing units or other switches.

W2 is a spare workstation for W1, i.e., when the workstation W1 is working, the workstation W2 is in standby mode, if W1 is failed, then it is replaced by W2. Connections between switches are bidirectional, i.e., each switch receives data from all of its neighbours and broadcasts data to all of them.

Figure 1 A data gathering and processing network



All components may fail in operation. Moreover, workstations may have a common cause failure. Failure rates are given in Table 1.

Table 1 Failure rates of components of the network

Component	*Failure rate*
Switch	10^{-4} 1/h
Workstation	10^{-5} 1/h
Processing unit	10^{-4} 1/h

This network is considered as robust if at least one of the processing units can send information to the plant. Since it is asymmetric, we want actually to assess the probability of each of the following three events:

- processing unit P1 cannot send data to the plant
- processing unit P2 cannot send data to the plant
- neither P1 nor P2 can send data to the plant.

We shall use this network as a red wire example both to present the new features of AltaRica 3.0 and to describe the different steps of the compilation of AltaRica 3.0 models into Fault Trees.

The reader has already noticed that this system contains loops since the information between switches can be transmitted both ways. As a consequence, the manual construction of a Fault Tree for each of the above top events is far from easy. It requires to analyse the various combinations of failures of components in order to determine if they lead to a total or partial loss of the data processing capacity.

3 AltaRica 3.0

AltaRica is an event-oriented modelling language, i.e., it assumes that the system under study changes of state when, and only when, an event occurs. The state of the system is described by means of variables. Changes of states are described by transitions. The occurrence of an event (the firing of a transition) updates the value of variables. Events can be associated with deterministic or stochastic delays so to obtain (stochastic) timed models. Models of components can be assembled into hierarchies. Variables are also used to propagate information through the network of components, i.e., inputs and outputs of blocks can be connected in various ways. AltaRica provides also a mechanism to synchronise transitions of different blocks.

Prior to AltaRica 3.0, two main versions of the language have been designed, which differ essentially in the way variables are updated after each transition firing. In the first version of the language, variables were updated by solving constraints (Point and Rauzy, 1999; Arnold et al., 2000). This mechanism, although very powerful, proved to be too resource consuming for industrial scale applications. Therefore a second version, AltaRica Data-Flow (Rauzy, A. (2002; Boiteau et al., 2006), has been designed in which variables are updated by propagating values in a fixed order (through data-flow equations). This order is determined at compile time, which imposes strong constraints on the way updating assertions are written and prevents notably to handle systems with loops. AltaRica 3.0 improves AltaRica Data-Flow into two directions:

1 Its semantics is based on a new underlying mathematical model: GTS (Rauzy, 2008b; Prosvirnova, T. and Rauzy, 2012). GTS makes it possible to handle systems with instant loops and to define acausal components, i.e., components in which the direction of the flow of information is determined at run time.

2 It provides new constructs to structure models.

3.1 Simple components

To start with, let us consider the following model.

```
domain ComponentState {WORKING, FAILED}
class NonRepairableComponent
    ComponentState s (init = WORKING);
    parameter Real lambda = 1.0e-5;
    event failure (delay = exponential (lambda));
transition
    failure: s == WORKING -> s := FAILED;
end
```

The class NonRepairableComponent describes, as its name indicates, a generic non-repairable component. The state of the component is represented by a state variable s. Its initial value (WORKING) is defined by means of the attribute init. The component changes from state WORKING to state FAILED when the event failure occurs. This event is associated with a stochastic delay, namely an exponential probability distribution of parameter lambda. Unless specified otherwise, this parameter takes the value 10^{-5}.

3.2 Data-flow components

AltaRica distinguishes two types of variables: state variables that are introduced by the attribute init and flow variables that are introduced by the attribute reset. State variables are used to model states of components. They are modified only by transitions. Flow variables are used to model the information circulating into the network of components. Their value is updated after each transition firing by means of a set of equations gathered into an assertion.

The following class describes processing units. When they are working, processing units receive data in input, process them, and emit data as output. If they are failed, they do not process nor output any data.

```
class ProcessingUnit
    ComponentState s (init = WORKING);
    Boolean inFlow, outFlow (reset = false);
    parameter Real lambda = 1.0e-4;
    event failure (delay = exponential (lambda));
transition
    failure: s == WORKING -> s := FAILED;
assertion
```

```
    outFlow := s == WORKING and inFlow;
end
```

3.3 Inheritance

The above class is similar to the previous one (NonRepairableComponent) except for input and output flow (and the corresponding assertion) that have been added and the default value of lambda that has been modified.

As other object-oriented languages, AltaRica 3.0 provides a mechanism to extend and modify classes. The class ProcessingUnit can be better described by inheriting from the class NonRepairableComponent as follows.

```
class ProcessingUnit
    extends NonRepairableComponent (lambda = 1.0e-4);
    Boolean inFlow, outFlow (reset = false);
assertion
    outFlow := s == WORKING and inFlow;
end
```

The two definitions are strictly equivalent.

The model of the workstation is quite similar to the previous one. If the workstation is working then the variable outFlow must be true, otherwise it is false (the workstation does not send data to the network).

```
domain SpareComponentState {STANDBY, WORKING, FAILED}
class Workstation
    SpareComponentState s (init = STANDBY);
    Boolean outFlow (reset = false);
    event start;
    event failure(delay = exponential(lambda));
    parameter Real lambda = 1.0e-5;
transition
    start: s == STANDBY -> s := WORKING;
    failure: s == WORKING -> s := FAILED;
assertion
    outFlow := s == WORKING;
end
```

Also to be able to represent the fact that the workstation W2 is in standby mode when the workstation W1 is working, we define a new domain SpareComponentState and two transitions start and failure.

3.4 Acausal components

The model for switches is more interesting for it is acausal, conversely to the previous models that are directional (data-flow).

```
class Switch extends NonRepairableComponent;
    Boolean leftFlow, rightFlow (reset = false);
    Boolean inFlow, outFlow (reset = false);
assertion
    if s == WORKING then {
      leftFlow: = rightFlow or inFlow;
      rightFlow: = leftFlow or inFlow;
      }
     outFlow: = (s == WORKING) and
      (leftFlow or rightFlow or inFlow);
end
```

The above model asserts that, if the switch is working, then flows linking the switch to its neighbours are all true as soon as one of them is true. If the switch is failed, then there is no relation amongst them.

3.5 Hierarchical models

AltaRica 3.0 is a prototype-oriented modelling language (see, e.g., Noble et al., 1999, for a discussion on objects versus prototypes). Prototype orientation makes it possible to separate the knowledge into two distinct spaces:

- The stabilised knowledge, incorporated into libraries of on-the-shelf modelling components, which are declared as classes.
- The sandbox in which the system under study is modelled. In the sandbox, many components (including the system itself) are unique. It would be therefore rather artificial to describe them by classes. Blocks, i.e., prototypes, are used instead.

This separation of knowledge spaces is inspired from Hatchuel et al. (2002) C-K theory. There are several differences between blocks and classes but describing them would go to far for the purpose of the present article.

Workstation, switch and processing unit are on-the-shelf components: they are represented in AltaRica 3.0 by classes. The network depicted in Figure 1 is unique, it can be represented by means of a block, named Network. This block contains instances of classes Workstation, ProcessingUnit and Switch. It is represented in Figure 2.

Figure 2 AltaRica 3.0 model of the data gathering and processing network

```
block Network
  Workstation W1, W3;
  Workstation W2(s.init = STANDBY);
  Switch SW1, SW2, SW3, SW4, SW5, SW6;
  ProcessingUnit P1, P2;

  parameter Real lambda = 5e-6;
  event ccf(delay = exponential(lambda));
  event W1_failure(delay = exponential(W1.lambda));

  observer Boolean P1P2failed =
    not P1.outFlow and not P2.outFlow;
  observer Boolean P1failed = not P1.outFlow;
  observer Boolean P2failed = not P2.outFlow;

transition
  W1_failure: !W1.failure & ?W2.start;
  ccf: ?W1_failure & ?W2.failure & ?W3.failure;
  hide W1.failure, W2.start;

assertion
  SW1.rightFlow :=: SW2.leftFlow;
  SW2.rightFlow :=: SW3.leftFlow;
  SW3.rightFlow :=: SW4.leftFlow;
  SW4.rightFlow :=: SW5.leftFlow;
  SW5.rightFlow :=: SW6.leftFlow;
  SW6.rightFlow :=: SW1.leftFlow;
  SW1.inFlow := W1.outFlow or W2.outFlow;
  SW3.inFlow := W3.outFlow;
  P1.inFlow := SW5.outFlow;
  P2.inFlow := SW4.outFlow;
end
```

3.5.1 Observers

Observers are like flow variables, except that they cannot be used in transitions and assertions. They are quantities to be observed by the assessment tools.

3.5.2 Synchronisations

Synchronisations are used to compel several events to occur at the same time. In the example given above in the initial state the workstation W1 is working and the workstation W2 is in standby mode. If W1 is failed, it is replaced by W2. This constraint is described by means of a synchronisation. A new event W1_failure is declared; it synchronises the events W1.failure and W2.start.

W1_failure: !W1.failure & ?W2.start;

This synchronisation creates a new transition:

```
W1_failure: W1.s == WORKING -> {W1.s := FAILED;
    if W2.s == STANDBY then W2.s := WORKING;}
```

The event W1.failure is prefixed with an exclamation mark (!). This modality indicates that the individual transition W1.failure is mandatory for the synchronised transition to be fired. On the other hand the event W2.start is prefixed with a question mark (?). This modality makes it possible to synchronise transitions only when they are fireable. In this case, the event W1_failure can occur even if the workstation W2 is not in standby mode.

The event ccf represents the common cause failure of the three workstations; it synchronises the events W1_failure, W2.failure and W3.failure. The corresponding transition is as follows.

```
ccf: W1.s == WORKING or W2.s == WORKING or W3.s == WORKING ->
{   if W1.s == WORKING then W1.s: = FAILED;
    if W1.s == WORKING and W2.s == STANDBY then W2.s: = WORKING;
    if W2.s == WORKING then W2.s: = FAILED;
    if W3.s == WORKING then W3.s: = FAILED;}
```

A synchronisation can involve any number of events. Events involved in a synchronisation continue to exist individually. In case of a common cause failure, the synchronised events represent individual failures of components and exist independently of the event ccf. However, in some situations events involved in a synchronisation should not occur individually. For example, the event W2.start should not occur if W1 is not failed. In this case it is possible to hide synchronised events in order to prevent them to occur individually. Hidden events and transitions are just removed from the model. Since all the instructions are executed in parallel (for more details see Section 4.1), the operation of synchronisation is commutative and associative. The reader interested in a more detailed description of synchronisations should read reference (Rauzy, 2013).

3.5.3 Assertion

Assertion represents connections between different components. The operator :=: represents bidirectional flows: data can circulate both ways between switches. The direction of the flow is determined at run time for it depends on the (global) state of the system.

Let us now examine how the flow propagation mechanism works on a couple of configurations. Assume first that the workstations W1 and W3 and the switch SW6 are failed. Since the workstation W2 is working, the flow variable W2.outFlow must be true. This value propagates thanks to the assertion to P1.outFlow and P2.outFlow via flows of switches SW1, SW2, SW3, SW4 and SW5 in order. Now if the switch SW2 is failed rather than the switch SW6, then the value true propagates from W2.outFlow to P1.outFlow and P2.outFlow via the flow of SW1, SW6, SW5 and SW4 in order. In the first case, the flow circulates from SW4 to SW5. In the second case, it circulates in the reverse direction.

This example illustrates the power of the AltaRica 3.0 language, which in few lines of clear code can model quite complex phenomena.

3.6 Flattening

Each hierarchical AltaRica 3.0 model can be flattened into a unique GTS, i.e., a model containing only declarations of variables, events, parameters, observers, transitions and assertions. Flattening of a block or a class is a purely syntactic operation. It works bottom up, i.e., that variables, events, observers, parameters, transitions and assertions of the inner block (or instance of class) are copied in the outer block (or instance of class) prefixing identifiers with the name of the inner block. Acausal connection, i.e., an instruction of the form $v :=: w$, where v and w are flow variables, is equivalent to two assignments: $\{v: = w; w: = v\}$. The reader interested in a detailed description of flattening should read reference (Rauzy, 2013).

The model obtained by flattening the block network is given in Figure 3. As for most of the assessment algorithms, flattening is the first step of the compilation of AltaRica 3.0 models into Fault Trees.

4 Compilation algorithm

The automatic generation of a Fault Tree from AltaRica 3.0 models is performed according to the following ideas:

- The basic events of the Fault Tree are the events of the AltaRica 3.0 model.
- There is (at least) an intermediate event for each pair (variable, value) of the AltaRica 3.0 model.
- For each minimal cut set of the Fault Tree rooted by an intermediate event (variable, value), there exists at least one sequence of transitions in the AltaRica 3.0 model labelled by the events of the cut set that ends up in a state where this variable takes this value. Moreover, this sequence is minimal in the sense that no strict subset of the minimal cut sets can label a sequence of transitions ending up in a state where this variable takes this value.

The generated Fault Tree is then exported in Open-PSA format (Hibti et al., 2012) and can be assessed with any calculation engine supporting this format, in order to calculate minimal cut sets, probabilities of the top events, importance factors, etc. The whole assessment process is illustrated in Figure 4.

The algorithm works in five steps (see Figure 5):

Step 1 The AltaRica 3.0 model is flattened into a GTS.

Step 2 The obtained GTS is partitioned into independent GTSs plus an independent assertion.

Step 3 Reachability graphs of each independent GTS are calculated.

Step 4 Each reachability graph is separately compiled into Boolean equations.

Step 5 The independent assertion is compiled into Boolean equations.

The first step has been (informally) described in the previous section. In this section, we shall describe the four following steps. In order to do so, we need to introduce GTS more formally.

Figure 3 Flattened network block

```
block NetworkSystemFlat
  SpareComponentState W1.s, W3.s(init = WORKING);
  SpareComponentState W2.s(init = STANDBY);
  ComponentState SW1.s, …, SW6.s(init = WORKING);
  ComponentState P1.s, P2.s(init = WORKING);
  Boolean P1.inFlow, …, P2.outFlow(reset = FALSE);
  Boolean SW1.leftFlow, …, SW6.outFlow(reset = FALSE);
  ⋮
  parameter Real W1.lambda = 0.0001;
  parameter Real W2.lambda = 0.0001;
  ⋮
  event W1_failure(delay = exponential(W1.lambda));
  event W2.failure(delay = exponential(W2.lambda));
  ⋮
  observer Boolean P1P2failed =
    not P1.outFlow and not P2.outFlow;
  observer Boolean P1failed = not P1.outFlow;
  observer Boolean P2failed = not P2.outFlow;
transition
  W1_failure: W1.s == WORKING -> {W1.s := FAILED;
      if W2.s == STANDBY then W2.s := WORKING;}
  W2.failure: W2.s == WORKING -> W2.s := FAILED;
  W3.failure: W3.s == WORKING -> W3.s := FAILED;
  ⋮
assertion
  if SW2.s == WORKING then
    SW2.leftFlow := SW2.rightFlow or SW2.inFlow;
  if SW2.s == WORKING then
    SW2.rightFlow := SW2.leftFlow or SW2.inFlow;
  ⋮
  SW1.rightFlow := SW2.leftFlow;
  SW2.leftFlow := SW1.rightFlow;
  ⋮
  P1.inFlow := SW6.outFlow;
  P2.inFlow := SW4.outFlow;
end
```

Figure 4 Fault tree analysis with AltaRica 3.0 models

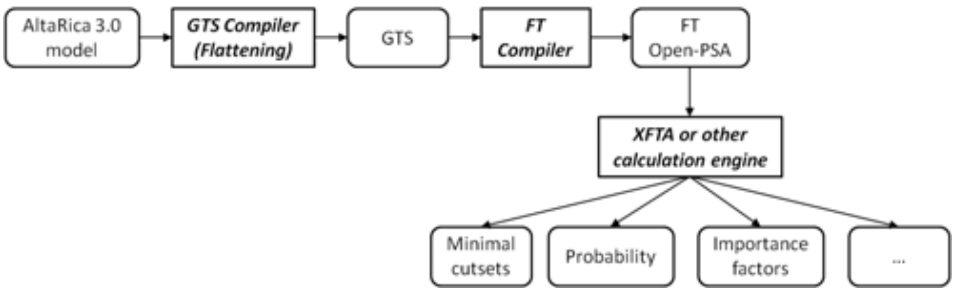
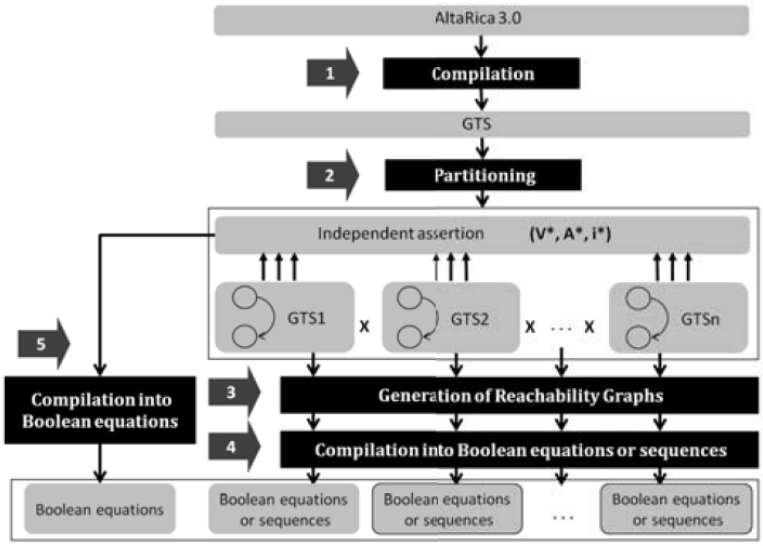


Figure 5 Compilation of AltaRica 3.0 models into Fault Trees



4.1 Formal definition and semantics of GTS

A GTS G is a quintuple $\langle V, E, T, A, \iota \rangle$, where:

- $V = S \uplus F$ is a set of variables, divided into disjoint sets: a set S of state variables and a set F of flow variables.
- E is a set of events.
- T is a set of transitions. A transition is a triple $t = \langle e, G, P \rangle$, where e is an event from E, G is a Boolean expression built over variables from V called the guard of the transition, and P is an instruction built over V called the action or the post-condition of the transition.
- A is an assertion (i.e., an instruction built over V).
- ι is the initial (or default) assignment of variables of V.

A GTS $G = \langle V, E, T, A, \iota \rangle$ is an implicit representation of a labelled Kripke structure, i.e., a graph $\Gamma = (\Sigma, \Theta)$, where

- Σ is a set of states, i.e., nodes labelled by the variable assignments
- Θ is a set of transitions, i.e., edges labelled by the events from E.

GTS are built on the following instructions:

- *Empty instruction:* 'skip' is an instruction.
- *Assignment:* If v is a variable and *Exp* an expression, then 'v: = *Exp*' is an instruction.

- *Conditional assignment:* If C is a Boolean expression, I is an instruction, then 'if C then I' is an instruction.
- *Parallel composition:* If I_1 and I_2 are two instructions, then so is 'I_1, I_2'.

We shall consider two types of instructions:

- Actions, i.e., instructions in which left members of assignments are only state variables.
- Assertions, i.e., instructions in which left members of assignments are only flow variables.

Let σ be a variable assignment just before the firing of the transition $t = \langle e, G, P \rangle$. Applying the instruction P to the variable assignment σ consists in calculating a new variable assignment τ according to the rules given in a Structural Operational Semantics style in Table A1 in Appendix. It is important to note that right hand side of assignments and conditions of conditional instructions are evaluated in the variable assignment σ. As a consequence, the result does not depend on the order in which instructions of a block are applied: they are executed in parallel. We denote by $Update(P, \sigma)$ a function that:

- extends a given partial variable assignment τ by means of a total variable assignment σ from the instruction P according to the rules given in Table A1 and starting with $\tau = \emptyset$
- completes τ by setting $\tau(v) = \sigma(v)$ for all state variables that are not given a value by the previous step.

Let A be an assertion, ι be an initial or a default variable assignment and τ the variable assignment obtained after the application of the action of a transition $\tau = Update(P, \sigma)$. Let denote by $Propagate(A, \iota, \tau)$ a function that:

- Extends a partial variable assignment τ by the instruction A according to the rules given in Table A2 in Appendix.
- Completes τ by setting all unassigned variables to its default values $\forall v \in F : \tau(v) = ?/\ \tau(v) = \iota(v)$.
- Verifies that all assignments of A are satisfied. If there is at least an assignment that is not satisfied then an error is raised.

Assume that the transition t is fireable in the state σ, i.e., the guard G is verified in σ. Then, the firing of the transition transforms σ into the assignment $Fire(e: G \rightarrow P, A, \iota, \sigma)$ defined as follows:

$$Fire\left(e : G \rightarrow P, A, \iota, \sigma\right) = Propagate\left(A, \iota, Update(P, \sigma)\right)$$

The labelled Kripke structure $\Gamma = (\Sigma, \Theta)$ is constructed as follows:

1 The initial state σ_0 is obtained from the application of the assertion A to the default or initial variable assignment $\iota : \sigma_0 = Propagate(A, \iota, \iota) \in \Sigma$.

2 If $\sigma \in \Sigma$ and $\exists t = \langle e, G, P \rangle \in T$, such that the guard G is verified in σ then the state $\tau = Fire(P, A, \iota, \sigma) \in \Sigma$ and the transition $(\sigma, e, \tau) \in \Theta$.

4.2 Compilation of labelled Kripke structures into Boolean formulae

From now, we assume that a GTS $G = \langle V, E, T, A, \iota \rangle$ describes a system that may fail. The graph $\Gamma = (\Sigma, \Theta)$ is the reachability graph of *G*. The initial state, or initial variable assignment, $\sigma_0 \in \Sigma$ represents the nominal state of the system. Events from *E* represent failures of system components. Some states (variables assignments) $\sigma \in \Sigma$ represent failure states. Paths from σ_0 to these states represent scenarios of failure.

Let us consider that the set of events *E* is an alphabet. Then, let denote by L_E a language built over the alphabet *E* and by ϵ an empty word. First of all, we search for all paths π from the initial state σ_0 to each state of the graph σ and associate a word $\phi(\sigma)$ from L_E to each state $\sigma \in \Sigma_i$ of the reachability graph Γ. This word $\phi(\sigma)$ is calculated as follows:

1 $\phi(\sigma_0) = \epsilon$

2 $\phi(\sigma) \sum_{\sigma_k : (\sigma_k, e_k, \sigma) \in \Theta} \phi(\sigma_k) \otimes e_k,$

where the operators Σ and $\otimes$ can be interpreted in different ways.

In case of the generation of critical sequences of events the operator $\otimes$ is interpreted as a concatenation of sequences and the operator Σ denotes sets of sequences. We do not present this case since it is out of the scope of this paper.

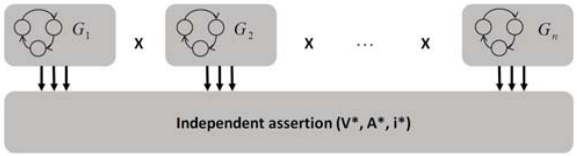
In case of the compilation into Fault Trees, the operator Σ is interpreted as a disjunction and the operator $\otimes$ as a conjunction. The algorithm captures failure scenarios into a set of Boolean equations. It produces a Boolean formula $\phi_{(v,c)}$ for each pair (*v*, *c*), where *v* is a variable from *V* and *c* is its value, $c \in dom(v)$, such that the variables of $\phi_{(v,c)}$ are events from *E*.

Due to the exponential blow up of the number of nodes of the reachability graph, it is not possible to directly generate the reachability graph and to compile it into Boolean formulae. Means should be found to take advantage of the independence of subsystems. Thus, before proceeding to the generation of the reachability graph, the model should by partitioned into independent parts.

4.3 GTS partitioning

Partitioning is a key point of the algorithm that ensures its efficiency. In practice, components of a system fail in general in a relatively independent way. In that case a partitioning is possible. Partitioning of a GTS $G = \langle V, E, T, A, \iota \rangle$ consists in representing *G* in the following way (see Figure 6 as an illustration):

Figure 6 Partitioning of GTS



$$G = G_1 \times G_2 \times \ldots \times G_n \uplus \langle V^*, A^*, \iota^* \rangle,$$

where

- $G_i = \langle V_i, E_i, T_i, A_i, \iota_i \rangle$ where are independent GTS, i.e., they do not share any variables and any events, $V_i \bigcap V_j = \varnothing \forall i \neq j$ and $E_i \bigcap E_j = \varnothing \forall i \neq j$,
- $\langle V^*, A^*, \iota^* \rangle$ is an independent assertion, also called a glue.

such that

- $V = V_1 \uplus V_2 \uplus \ldots \uplus V_n \uplus V^*$
- $E = E_1 \uplus E_2 \uplus \ldots \uplus E_n$
- $T = T_1 \uplus T_2 \uplus \ldots \uplus T_n$
- $A = A_1; A_2; \ldots; A_n; A^*$, where the operator; denotes the parallel composition of instructions
- $\iota = \iota_1 \circ \iota_2 \circ \ldots \circ \iota_n \circ \iota^*$, where the operator $\circ$ denotes the composition of functions.

The last part (the glue) does not contain any behaviour. Variables in V^* are only flow variables, they depend on state variables and flow variables of independent GTSs G_i. A similar idea can be found in Gössler and Sifakis (2005).

To partition a GTS one need to analyse dependencies between its transitions in order to divide them into independent groups. To do that, variables involved in each transition, i.e., variables used in the guard and in the action of the transition, should be considered.

Let denote by *var*(*Exp*) variables used in the expression *E*, and by *var*(*I*) variables used in the instruction *I*. Typically,

- if *I* is an empty instruction 'skip', then var(*I*) = Ø
- if *I* is in the form '*v*: = *Exp*', $\text{var}(I) = v \cup \text{var}(Exp)$
- if *I* is in the form 'if *Exp* then *J*', $\text{var}(I) = \text{var}(Exp) \cup \text{var}(J)$
- if *I* is in the form 'I_1; I_2', then $\text{var}(I) = \text{var}(I_1) \cup \text{var}(I_2)$.

Definition 4.1: Let *v* and *w* be two variables and *I* an instruction. We say that *v* depends immediately on *w* if the following holds:

- *I* is in the form '*v*: = *Exp*' and $w \in \text{var}(Exp)$
- *I* is in the form 'if *Exp* then *J*', where *J* is an instruction, and $w \in \text{var}(Exp)$ or *v* depends immediately on *w* in *J*
- *I* is in the form 'I_1; I_2' and *v* depends immediately on *w* either in I_1 or in I_2.

Definition 4.2: Let *v* and *w* be two variables and *I* an instruction. We say that *v* depends on *w* in the instruction *I*, if there is a variable *u* such that *v* depends immediately on *u* in *I* and *u* depends on *w* in *I*.

Let denote by var(*t*) variables involved in the transition $t = \langle e, G, P \rangle$:

$$\mathrm{var}(t) = \mathrm{var}(G) \cup \mathrm{var}(P) \cup V',$$

where V' are variables, such that variables from $\mathrm{var}(G) \cup \mathrm{var}(P)$ depend on them via the assertion A:

$$V' = \left\{ v \in V : \exists u \in \mathrm{var}(G) \cup \mathrm{var}(P) \middle| u \text{ depends on } v \text{ in } A \right\}$$

Definition 4.3: Let t_1 and t_2 be two transitions. We say that t_1 and t_2 are independent if $\mathrm{var}(t_1) \cap \mathrm{var}(t_2) = \varnothing$.

The last relation enables to divide transitions and, as a consequence events and variables, into independent sets T_i, E_i, V_i. The assertion A can also be divided into independent parts A_i. The remaining flow variables and instructions from A constitute the independent assertion $< V^*, A^*, \iota^* >$.

Figure 7 Partitioned network

```
block Part1
  ComponentState W1.s, W3.s (init = WORKING);
  ComponentState W2.s (init = STANDBY);
  Boolean W1.outFlow,…, SW3.inFlow(reset = FALSE);
  ⋮
  event W1_failure(delay = exponential(W1.lambda));
  event W2.failure(delay = exponential(W2.lambda));
  event W3.failure(delay = exponential(W3.lambda));
  event ccf(delay = exponential(lambda));
transition
  W1_failure: W1.s == WORKING ->
    { W1.s = FAILED;
      if ( W2.s == STANDBY ) then W2.s = WORKING; }
  W2.failure: W2.s == WORKING -> W2.s := FAILED;
  W3.failure: W3.s == WORKING -> W3.s := FAILED;
  ccf : W1.s==WORKING or W2.s==WORKING or W3.s==WORKING ->
    { if ( W1.s == WORKING ) then W1.s = FAILED;
    if W1.s == WORKING and W2.s == STANDBY then W2.s = WORKING;
    if W2.s == WORKING then W2.s = FAILED;
    if W3.s == WORKING then W3.s = FAILED;}
assertion
  W1.outFlow := W1.s == WORKING;
  W2.outFlow := W2.s == WORKING;
  W3.outFlow := W3.s == WORKING;
  SW1.inFlow := W1.outFlow or W2.outFlow;
  SW3.inFlow := W3.outFlow;
end
  ⋮
block Part9
  ComponentState P2.s (init = WORKING);
  parameter Real P2.lambda = 1.0e-4;
  event P2.failure(delay = exponential(P2.lambda));
transition
  P2.failure: P2.s == WORKING -> P2.s := FAILED;
end
block IndependentAssertion
  Boolean SW2.leftFlow (reset = FALSE);
  Boolean SW2.rightFlow (reset = FALSE);
  ⋮
assertion
  if SW2.s == WORKING then
    SW2.leftFlow := SW2.rightFlow or SW2.inFlow;
  if SW2.s == WORKING then
    SW2.rightFlow := SW2.leftFlow or SW2.inFlow;
  ⋮
  SW1.rightFlow := SW2.leftFlow;
  SW2.leftFlow := SW1.rightFlow;
end
```

In the model of the network system, the transition ccf involves the variables W1.s, W2.s and W3.s. Since the transitions ccf, W1_failure, W2.failure, W3.failure share variables, used in their guards and actions, they belong to the same partition. Since flow variables Wi.outFlow, i = 1..3, and SWi.inFlow, i = 1, 3 do not depend on variables from other partitions, they belong to this partition, and it also the case of the corresponding assertions. Other transitions SW1.failure, …, SW6.failure, P1.failure, P2.failure are independent from each other and form distinct partitions. The partitioned network system contains nine independent GTSs and an independent assertion. They are represented by blocks in Figure 7.

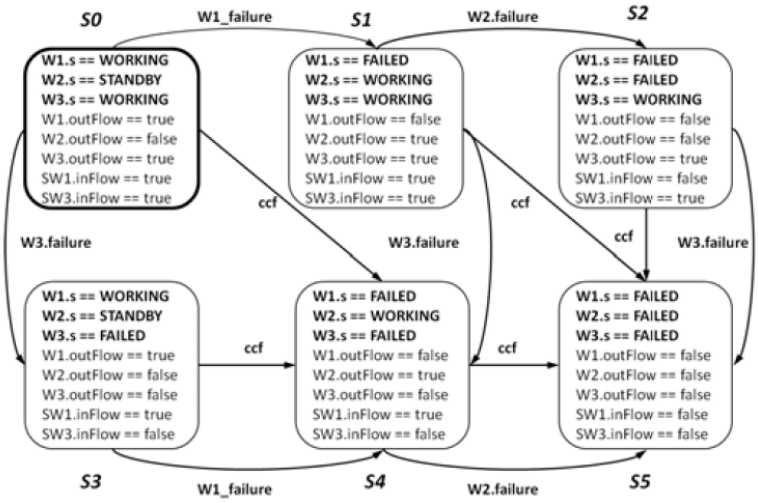
Remark 1: Partitioning is a purely syntactic operation. Note that generally the partitioned GTS does not correspond to the structure of the initial AltaRica 3.0 model. It is particularly the case of the example given above. Only in case of completely independent components the partitioned model corresponds to the structure of the AltaRica 3.0 model.

4.4 Reachability graph generation

For each independent GTS G its reachability graph $\Gamma = (\Sigma, \Theta)$ is constructed according to the definition given in Section 4.1. Starting from the initial state $\sigma_0 = Propagate(A, \iota, \iota)$, calculated using the initial variable assignment ι and the assertion A, other states are discovered by firing step by step all transitions, fireable in the current state: if $\sigma \in \Sigma$ and there is a transition $t = \langle e, G, P \rangle$ which is fireable in σ, then $\tau = Fire(t, A, \iota, \sigma) \in \Sigma$ and $(\sigma, e, \tau) \in \Theta$.

The reachability graph of the block Part1 is depicted in Figure 8. It contains six nodes (or states) S0, …, S5, labelled by variable assignments. Its transitions are labelled by the events of the block Part1: ccf, W1_failure, W2.failure and W3.failure. We shall use this graph in order to illustrate how a reachability graph is compiled into Boolean formulae.

Figure 8 Reachability graph of spare workstations



Reachability graphs for other partitions of the network system are trivial. They contain only two states (when state variable s of each component is equal to WORKING and when it is equal to FAILED) and one transition labelled by the event failure of the corresponding component.

4.5 Compilation of the independent reachability graphs into Boolean formulae

Consider an independent GTS G and its reachability graph $\Gamma = (\Sigma, \Theta)$. The principle of compilation of Γ into Boolean formulae is given in Section 4.2. First, we search for all paths π from the initial state σ_0 to each state of the graph σ. In order to avoid conflicts raised by the composition of components (for more details, see Rauzy, 2002) we need to consider not only events occurring along the path π but also those that do not. Finally, the algorithm works as follows:

1 Associate with each state $\sigma \in \Sigma$ a set of Boolean vectors P_σ: $E \rightarrow \{TRUE, FALSE\}$, with $P_\sigma(e_k)$ = TRUE if the event e_k occurs along the path π from σ_0 to σ.

2 Then for each couple (σ, P_σ) associate a Boolean equation $\phi_{(\sigma,P_\sigma)}$, obtained as follows:

$$\phi_{(\sigma,P_\sigma)} = \bigwedge_{e_k \in E:P_\sigma(e_k)=TRUE} e_k \bigwedge_{e_j \in E:P_\sigma(e_j)=FALSE} \overline{e}_j$$

3 For each state σ associate a Boolean equation obtained as follows:

$$\phi_\sigma = \bigvee_{(\sigma,P_\sigma)} \phi_{(\sigma,P_\sigma)}$$

4 Finally, for each couple (v, c), where $v \in V$ is a variable and $c \in dom(v)$ is its value, associate a Boolean equation as follows:

$$\phi_{(v,c)} = \bigvee_{\sigma \in \Sigma:\sigma(v)=c} \phi_\sigma$$

In order to illustrate this algorithm consider the reachability graph given in Figure 8. Boolean equations associated with each node (or state) of this graph (step 3 of the algorithm) are as follows:

- $\phi_{S0} = \overline{ccf} \wedge \overline{W1\_failure} \wedge \overline{W2.failure} \wedge \overline{W3.failure}$
- $\phi_{S1} = \overline{ccf} \wedge W1\_failure \wedge \overline{W2.failure} \wedge \overline{W3.failure}$
- $\phi_{S2} = \overline{ccf} \wedge W1\_failure \wedge W2.failure \wedge \overline{W3.failure}$
- $\phi_{S3} = \overline{ccf} \wedge W1\_failure \wedge \overline{W2.failure} \wedge W3.failure$
- $\phi_{S4} = \overline{ccf} \wedge W1\_failure \wedge \overline{W2.failure} \wedge W3.failure \vee$
  $ccf \wedge \overline{W1\_failure} \wedge \overline{W2.failure} \wedge W3.failure \vee$
  $ccf \wedge \overline{W1\_failure} \wedge \overline{W2.failure} \wedge \overline{W3.failure}$

$$\phi_{S5} = ccf \wedge \overline{W1\_failure} \wedge \overline{W2.failure} \wedge \overline{W3.failure} \vee$$
$$ccf \wedge \overline{W1\_failure} \wedge W2.failure \wedge \overline{W3.failure} \vee$$
$$ccf \wedge \overline{W1\_failure} \wedge \overline{W2.failure} \wedge W3.failure \vee$$
$$ccf \wedge \overline{W1\_failure} \wedge W2.failure \wedge W3.failure \vee$$
$$ccf \wedge W1\_failure \wedge \overline{W2.failure} \wedge \overline{W3.failure} \vee$$
$$ccf \wedge W1\_failure \wedge W2.failure \wedge \overline{W3.failure} \vee$$
$$ccf \wedge W1\_failure \wedge \overline{W2.failure} \wedge W3.failure \vee$$
$$\overline{ccf} \wedge W1\_failure \wedge W2.failure \wedge W3.failure$$

Then, Boolean equations for each couple variable and its value (v, c) (step 4 of the algorithm) are calculated:

$$\phi_{(W1.s,WORKING)} = \phi_{S0} \vee \phi_{S3} \qquad \phi_{(W1.s,FAILED)} = \phi_{S1} \vee \phi_{S2} \vee \phi_{S4} \vee \phi_{S5}$$

$$\phi_{(W2.s,STANDBY)} = \phi_{S0} \vee \phi_{S3} \qquad \phi_{(W2.s,WORKING)} = \phi_{S1} \vee \phi_{S4}$$

$$\phi_{(W2.s,FAILED)} = \phi_{S2} \vee \phi_{S5}$$

$$\phi_{(W3.s,WORKING)} = \phi_{S0} \vee \phi_{S1} \vee \phi_{S2} \qquad \phi_{(W3.s,FAILED)} = \phi_{S3} \vee \phi_{S4} \vee \phi_{S5}$$

$$\phi_{(W1.s,outFlow,true)} = \phi_{S0} \vee \phi_{S2} \qquad \phi_{(W1.outFlow,false)} = \phi_{S1} \vee \phi_{S3} \vee \phi_{S4} \vee \phi_{S5}$$

$$\phi_{(W2.s,outFlow,true)} = \phi_{S1} \vee \phi_{S3} \qquad \phi_{(W2.outFlow,false)} = \phi_{S0} \vee \phi_{S2} \vee \phi_{S4} \vee \phi_{S5}$$

$$\phi_{(W3.s,outFlow,true)} = \phi_{S0} \vee \phi_{S1} \vee \phi_{S4} \qquad \phi_{(W23outFlow,false)} = \phi_{S2} \vee \phi_{S3} \vee \phi_{S5}$$

$$\phi_{(SW1.inFlow,false)} = \phi_{S4} \vee \phi_{S5} \qquad \phi_{(SW1.inFlow,true)} = \phi_{S0} \vee \phi_{S1} \vee \phi_{S2} \vee \phi_{S3}$$

$$\phi_{(SW3.inFlow,true)} = \phi_{S0} \vee \phi_{S1} \vee \phi_{S4} \qquad \phi_{(SW3.inFlow,false)} = \phi_{S2} \vee \phi_{S3} \vee \phi_{S5}$$

Boolean equations generated from other reachability graphs are very simple. For example,

$$\phi_{(SW1.s,WORKING)} = \overline{SW1.failure}$$

$$\phi_{(SW1.s,FAILED)} = SW1.failure$$

4.6 Compilation of the assertion into Boolean formulae

Let denote by U the set of variables from independent GTSs:

$$U = V_1 \uplus V_2 \uplus \ldots \uplus V_n$$

In the previous step $\forall u \in U,\ c \in dom(u)$ a Boolean equation $\phi_{(u,c)}$ has been calculated.

The independent assertion $\langle V^*, A^*, \iota^* \rangle$ is transformed into a set of Boolean formulae in the following way. For each pair (f, q), where $f \in V^*$ is a flow variable and $q \in dom(f)$ is its value, a Boolean formula $\phi_{(f,q)}$ is constructed according to the

instructions in the assertion A^* and Boolean formulae $\{\phi_{(u,c)}, u \in U, c \in dom(u)\}$ obtained from the compilation of the independent GTSs.

In order to compile the assertion into Boolean formulae efficiently, one need to separate it into independent parts. The dependency relation between variables in the assertion A^* defines a dependency graph. This graph may contain cycles. The strongly connected components of this graph divide variables of A^* into sets and enable to decompose the assertion A^* into blocks of instructions A_i, i = 1 .. m, where m is the number of strongly connected components:

$$A^* = A_1^*; A_2^*; \ldots; A_m^*$$

Each block of instructions A_i^* is compiled into Boolean formulae recursively according to the algorithm described hereafter. Let denote by

- V_i^* – a set of variables labelling the vertices of the strongly connected component number *i*.
- A_i^* – an instruction that calculates the values of variables from V_i^*.
- ι_i^* – an initial assignment of variables from V_i^*.
- W_i^* – a set of variables such that variables from V_i^* depend on them in A_i^*.

Assume that $\forall w \in W_i^* dom(w)$ and $\{\phi_{(w,c)}, w \in W_i^*, c \in dom(w)\}$ has been calculated (either by the step 3 of the algorithm or by the previous step of the recursion). Then to compute $\{\phi_{(v,c)}, v \in V_i^*, c \in dom(v)\}$, one need

1 First, to compute the Cartesian product of the domains of variables from W_i^*, i.e., the set

$$\sum = \underset{w \in W_i^*}{\times} dom(w)$$

2 Then, for each assignment of variables from $W_i^* \sigma \in \Sigma$:

a to compute a Boolean equation associated with the variable assignment σ

$$\phi_\sigma = \underset{w \in W_i^*}{\wedge} \phi_{(w,\sigma(w))}$$

b to compute a partial variable assignment $\tau : V_i^* \cup W_i^* \to \mathcal{C}$ as follows:

$$\forall w \in W_i^* \tau(w) = \sigma(w)$$

c to complete the partial variable assignment τ by propagating the assertion A_i^*:

$$\tau = Propagate\left(A_i^*, \iota_i^*, \tau\right)$$

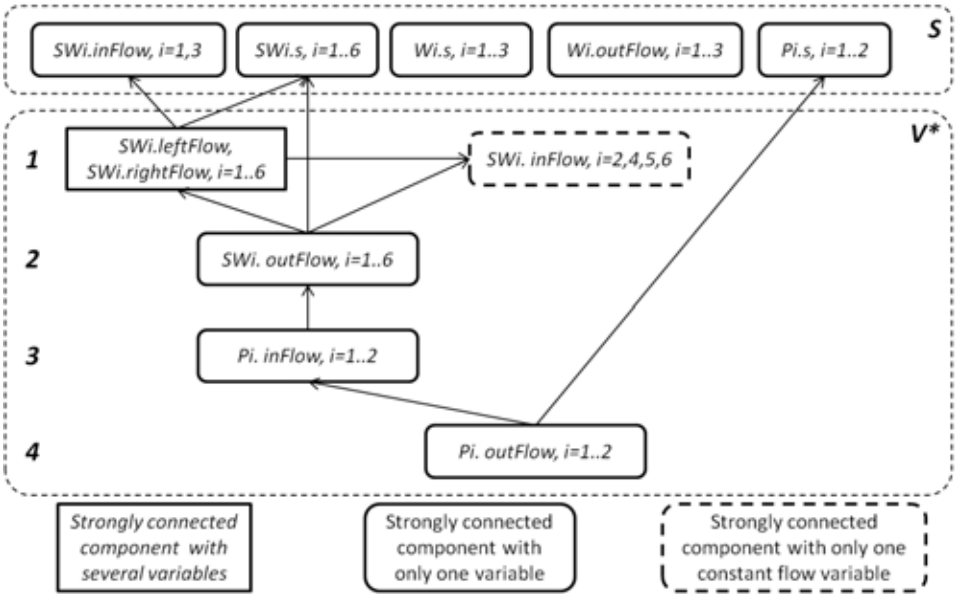
d to update Boolean equation associated with each couple (v, c), $v \in V_i^*$, such that $\tau(v) = c$, as follows:

$$\phi_{(v,c)} \leftarrow \phi_{(v,c)} \vee \phi_\sigma$$

Remark 2: In case of a data-flow assertion the dependency graph does not contain any cycles and the number of strongly connected components is equal to the number of variables and also to the number of the assignments (as long as each variable is assigned only once). Since the number of variables used in the right hand side of the assignments is never big, this operation is performed very efficiently.

Let us explain how this algorithm works for the independent assertion of the red wire example of this article. Boolean formulae for variables of the independent GTSs of the Network system $W_i.s$, i = 1..3, $W_i.outFlow$, i = 1..3, $SW_i.s$, i = 1..6, $SW_i.inFlow$, i = 1, 3 and $P_i.s$, i = 1, 2 have been calculated by the previous step of the algorithm. Variables of the independent assertion V^* (i.e., flow variables) depend on them via the assertion A^*. The dependency graph of A^* contains cycles, since acausal components have been used in the model. Strongly connected components of this dependency graph are represented in Figure 9.

Figure 9 Dependency graph of the independent assertion



Variables $SW_i.leftFlow$, $SW_i.rightFlow$, i = 1..6 belong to the same strongly connected component and depend on variables $SW_i.inFlow$, i = 1..6 and $SW_i.s$, $\underline{i}$ = 1..6 (see Figure 9). Let us denote this set of variables V_0^*. Boolean equations for variables from the same strongly connected component are calculated together.

1 We start with

$$\phi_{v,c)} = \varnothing \forall v \in V_0^*, c \in dom(v)$$

2 For each assignment σ of variables $SW_i.inFlow$, i = 1, 3, $SW_i.s$, i = 1..6

a first, the corresponding Boolean formula ϕ_σ is calculated

b then, values of variables from V_0^* are calculated according to the assertion A^* and the assignment σ

c finally, for each variable $v \in V_0^*$ and its value c the corresponding Boolean formula $\phi_{(v,c)}$ is updated as follows:

$$\phi_{(v,c)} \leftarrow \phi_{(v,c)} \vee \phi_\sigma$$

For example, consider a configuration where W1, W3 and SW6 are failed and all other components are working. In that case *SW*1.*inFlow* = *true* and *SW*3.*inFlow* = *false*. The corresponding Boolean equation ϕ_σ is as follows:

$$\phi_\sigma = \phi_{(SW1.inFlow,true)} \wedge \phi_{(SW3.inFlow,true)} \wedge \phi_{(SW6.s,FAILED)} \wedge \phi_{(SW1.s,WORKING)} \wedge \cdots \wedge \phi_{(SW5.s,WORKING)}$$

Since W2 is working, *W*2.*outFlow* = *true* and by propagation all other variables are equal to true. The following Boolean equations are generated:

$$\phi_{(SW1.leftFlow,true)} \leftarrow \phi_{(SW1.leftFlow,true)} \vee \phi_\sigma$$

$$\vdots$$

$$\phi_{(SW6.leftFlow,true)} \leftarrow \phi_{(SW6.leftFlow,true)} \vee \phi_\sigma$$

The same procedure is applied recursively for all strongly connected components of the dependency graph of A^*. Finally, we obtain a set of Boolean equations $\phi_{(v,c)}$, $v \in V^*$, $c \in dom(v)$. Along with Boolean equations obtained by the compilation of the independent reachability graphs, they encode a set of Fault Trees for the model of the network system.

4.7 Complexity analysis

In the following, we estimate the (time) complexity of each step of the algorithm depending on the size of the input model:

Step 1 *Flattening* mainly consists in model rewriting. The complexity of this operation is linear on the size of the AltaRica 3.0 model.

Step 2 *Partitioning* is a syntactical operation. The complexity of this step is linear on the size of the GTS model generated by the previous step.

Step 3 *Reachability graph generation* is exponential on the number of state variables of the model. But the partitioning of the model enables not to generate the reachability graph of the whole model. Assume that there are *n* Boolean state variables in the model. In the worst case (when there is only 1 part) the complexity is $O(2^n)$. In the best case (when all state variables belong to different parts) the complexity is $O(n)$.

Step 4 Compilation of the reachability graphs into Boolean formulae is linear on the size of the graph.

Step 5 The independent assertion is compiled symbolically into Boolean equations. The complexity of this operation depends on the number of strongly connected components of the dependency graph of A^*. Let's consider that the dependency graph has m strongly connected components. Note that in case of a data-flow assertion m is equal to the number of variables in V^*. The complexity of each step of the algorithm is exponential on the number of outgoing edges of each strongly connected component. But in general the number of outgoing edges (i.e., the number of variables used in the right hand side of the assignments and in the conditions) is never big and then can be considered as constant. Thus, the complexity can be considered as linear on the number of strongly connected components of the dependency graph of A^*. In the best case, when the assertion A^* is data-flow, the complexity is linear on the number of flow variables assigned in A^* ($O(m)$). In the worst case, there is only one strongly connected component ($m = 1$) and there are n outgoing edges, where n is the number of state variables of the model, the complexity is exponential on n and the advantage of the partitioning is lost.

4.8 Correctness

The correctness of the algorithm relies on the following theorems:

Theorem 1: Let $G = \langle V, E, T, A, \iota \rangle$ be a GTS, Γ its reachability graph and $\{\phi_{v,c)}, v \in V, c \in dom(v)\}$ a set of Boolean formulae generated from G. If $\{e_1, \ldots, e_k\}$, e_j, $j = 1 \ldots k, \in E$ is a minimal cut set of $\phi_{v,c)}$, then there is a sequence of transitions labelled by these events in the reachability graph Γ, such that $(\sigma_0, e_1, \sigma_1) \in \Theta, (\sigma_1, e_2, \sigma_2) \in \Theta, \ldots, (\sigma_{k-1}, e_k, \sigma_k) \in \Theta,$ and $\sigma_k(v) = c$.

Proof: By construction (see Sections 4.2 and 4.5) the algorithm explores sequences of transitions starting from σ_0 and ending up in a state σ_k, such that $\sigma_k(v) = c$ and for a particular sequence of transitions generates a conjunction of events labelling the transitions of the sequence and negation of events that do not occur in the sequence.

Theorem 2: Let $G = \langle V, E, T, A, \iota \rangle$ be a GTS, Γ its reachability graph and $\{\phi_{v,c)}, v \in V, c \in dom(v)\}$ a set of Boolean formulae generated from G. If there is a sequence of transitions in the reachability graph Γ, such that $(\sigma_0, e_1, \sigma_1) \in \Theta, (\sigma_1, e_2, \sigma_2) \in \Theta, \ldots, (\sigma_{k-1}, e_k, \sigma_k) \in \Theta$ and this sequence is minimal (in the sense that that no strict subset of events of this sequence can label a sequence of transitions ending up in a state where $v = c$), then the set of events, labelling the transitions in the sequence $\{e_1, \ldots, e_k\}$, $e_j, j = 1 \ldots k, \in E$ is a minimal cut set of $\phi_{v,c)}$.

Proof: Also by construction (see Sections 4.2 and 4.5).

Theorem 3: Let $G = G_1 \times G_2 \times \ldots \times G_n \uplus < V^*, A^*, \iota^* >$ be a partitioned GTS and let $\Gamma = (\Sigma, \Theta)$ be a reachability graph of G. Then

$$\Gamma = \left(\Gamma_1 \otimes \Gamma_2 \otimes \ldots \otimes \Gamma_n\right)\Big|_{<V^*, A^*, \iota^*>},$$

where

- $\otimes$ is a free product of reachability graphs
- $|_{<V^*,A^*,\iota^*>}$ is the extension of the reachability graph by the assertion A^*, which is calculated as follows.

Let $\Gamma' = \Gamma_1 \otimes \Gamma_2 \otimes \ldots \otimes \Gamma_n$ and $\Gamma' = (\Sigma', \Theta')$. Each variable assignment σ: $V \setminus V^* \rightarrow C$ is extended to V: $\tau = \sigma|V$. The assignment τ is calculated as follows:

1 $\forall v \in V \setminus V^* \tau(v) = \sigma(v)$

2 $\tau = Propagate(A^*, \iota^*, \tau)$

Proof: The proof is based on the fact that GTSs G_i are built over distinct sets of variables and transitions and variables from V^* are also distinct from variables of independent GTSs.

5 Experiments

The initial AltaRica 3.0 model of the network system contains 45 variables and 12 events. The complete reachability graph contains 1,536 states and 9,216 transitions. The partitioned model has nine parts: reachability graphs of eight parts are very similar and contain only two states, the last one contains six states.

For comparison, execution times of the program with and without partitioning for the red wire example of this article are given in Table 2. These results have been obtained on a laptop computer with a processor Intel Core i7, a 6 GB memory and running on Windows 7.

Table 2 Execution times of the program for the model of the network system

Model	*Without partitioning*	*With partitioning*
Network system	4.852 sec.	0.187 sec.

The top events are specified via the observers P1failed, P2failed, P1P2failed and their value true. Thus, several Fault Trees can be generated from a unique AltaRica 3.0 model. For technical reasons, the Fault Trees generated by the AltaRica 3.0 compiler are quite different from those an analyst would write. The minimal cut sets are however the expected ones. For instance, the minimal cut sets, their probabilities and contributions for the top event 'P_1 *cannot send data to the plant*', defined by the observer P1failed, are given in Table 3. The same results for the top events 'P_2 *cannot send data to the plant*' (observer P2failed) and '*Neither* P_1 *nor* P_2 *can send data to the plant*' (observer P1P2failed) are summarised in Tables 4 and 5 respectively.

Probabilities of the top events are calculated for the period from 0 to 1,000 hours. The results are presented in Figure 10.

The generated Fault Trees have been assessed with XFTA calculation engine.

Table 3 Minimal cut sets for the top event 'P_1 cannot send data to the plant'

Rank	*Order*	*Probability*	*Contribution*	*Minimal cut set*
1	1	0.0951626	0.409152	P1.failure
2	1	0.0951626	0.409152	SW5.failure
3	2	0.00905592	0.038936	SW3.failure SW6.failure
4	2	0.00905592	0.038936	SW4.failure SW6.failure
5	2	0.00905592	0.038936	SW1.failure SW4.failure
6	2	0.00905592	0.038936	SW1.failure SW3.failure
7	1	0.00498752	0.0214439	ccf
8	2	0.000946884	0.00416035	SW1.failure W3.failure
9	3	9.01079e-5	0.00038742	SW2.failure SW6.failure W3.failure
10	3	9.42165e-6	4.05085e-5	SW3.failure W1_failure W2.failure
11	3	9.85124e-7	4.23555e-6	W1_failure W2.failure W3.failure
12	4	8.96588e-7	3.85489e-6	SW2.failure SW4.failure W1_failure W2.failure

Table 4 Minimal cut sets for the top event 'P_2 cannot send data to the plant'

Rank	*Order*	*Probability*	*Contribution*	*Minimal cut set*
1	1	0.0951626	0.425559	P2.failure
2	1	0.0951626	0.425559	SW4.failure
3	2	0.00905592	0.0404973	SW3.failure SW5.failure
4	2	0.00905592	0.0404973	SW3.failure SW6.failure
5	2	0.00905592	0.0404973	SW1.failure SW3.failure
6	1	0.00498752	0.0223038	ccf
7	2	0.000946884	0.00423438	SW1.failure W3.failure
8	3	9.01079e-5	0.000402955	SW2.failure SW5.failure W3.failure
9	3	9.01079e-5	0.000402955	SW2.failure SW6.failure W3.failure
10	3	9.42165e-6	4.21328e-5	SW3.failure W1_failure W2.failure
11	3	9.85124e-7	4.40539e-6	W1_failure W2.failure W3.failure

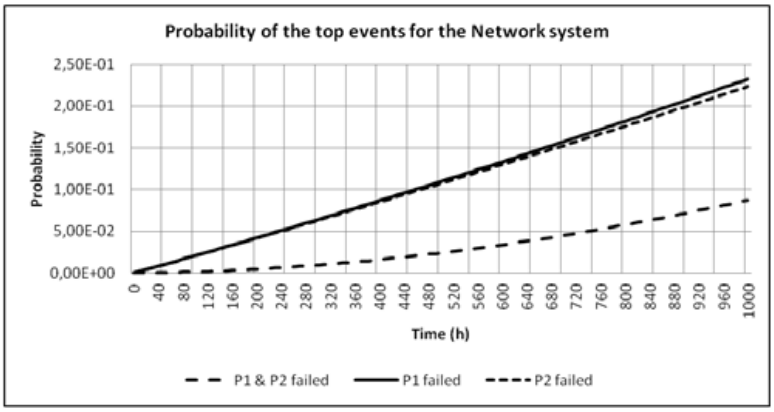
Table 5 Minimal cut sets for the top event 'neither P_1 nor P_2 can send data to the plant'

Rank	*Order*	*Probability*	*Contribution*	*Minimal cut set*
1	2	0.00905592	0.103344	P1.failure P2.failure
2	2	0.00905592	0.103344	P1.failure SW4.failure
3	2	0.00905592	0.103344	P2.failure SW5.failure
4	2	0.00905592	0.103344	SW4.failure SW6.failure
5	2	0.00905592	0.103344	SW1.failure SW4.failure
6	2	0.00905592	0.103344	SW4.failure SW5.failure
7	2	0.00905592	0.103344	SW3.failure SW5.failure
8	2	0.00905592	0.103344	SW1.failure SW3.failure

Table 5 Minimal cut sets for the top event 'neither P_1 nor P_2 can send data to the plant' (continued)

Rank	*Order*	*Probability*	*Contribution*	*Minimal cut set*
9	2	0.00905592	0.103344	SW3.failure SW6.failure
10	1	0.00498752	0.0569162	ccf
11	2	0.000946884	0.0108056	SW1.failure W3.failure
12	3	9.01079e-5	0.00102829	SW2.failure SW5.failure W3.failure
13	3	9.01079e-5	0.00102829	SW2.failure SW6.failure W3.failure
14	3	9.42165e-6	0.00010752	SW3.failure W1_failure W2.failure
15	3	9.85124e-7	1.1242e-5	W1_failure W2.failure W3.failure
16	4	8.96588e-7	1.02316e-5	SW2.failure SW4.failure W1_failure W2.failure

Figure 10 Probability of the top events



6 Related works

The automatic generation of Fault Trees from high level models is a wide domain of research. Different algorithms have been proposed in the literature. Most of the proposed algorithms can be divided into two groups:

- algorithms based on backward analysis
- algorithms based on fault injection.

6.1 Algorithms based on backward analysis

In this category most of the proposed approaches rely on various extensions of reliability block diagrams (see, e.g., Majdara and Wakabayashi, 2009). Basic blocks of the diagram carry out both failures and inputs/outputs relations. The flow circulating in the diagram is analysed backward to generate the Fault Tree. This idea stems from artificial intelligence tools such as assumption-based truth maintenance systems (de Kleer, 1986).

A similar idea is used in the HiP-HOPS workbench (Pasquini et al., 1999; Papadopoulos and Maruhn, 2001). This workbench enables to add reliability data to models imported from different modelling tools: MATLAB/Simulink, Eclipse-based UML tools, etc., and then to automatically generate Fault Trees and FMEA tables. The underlying formalism of Hip-HOPS is also an extension of reliability block diagrams, in which the system is described by hierarchies of blocks and the outputs of the blocks are written as a discrete function of internal failures and inputs. AltaRica 3.0 generalises this kind of models and the algorithm described in this article is very efficient on them.

The same principle is used in Joshi et al. (2007), where Fault Trees are automatically generated from AADL models.

6.2 Algorithms based on fault injection

In this case the model is simulated step by step in order to discover sequences of events leading from the nominal state to a failure state. Then, the generated Fault Tree is just a disjunction over all found sequences of conjunctions of events involved in each sequence. This approach implies to enumerate all combinations of failure events and to test them.

In this category, we can find the algorithm used in KB3 workbench (Bouissou, 2005), developed by EDF R&D, where Fault Trees are automatically generated from Figaro models (Bouissou et al., 1991). The proposed algorithm is based on the exploration of all possible combinations of failure events in order to determine if they lead to the system failure. Different truncation criteria are applied, such as the maximum number of events in a sequence, the probability of the sequence, etc.

The same principle is used in Bozzano et al. (2007), where the authors use NuSMV for Fault Tree analysis and apply different symbolic model-checking techniques in order to improve the efficiency of the algorithm.

In Bozzano et al. (2011), the authors translate AltaRica Data-Flow models, created within Cecilia OCAS workbench, into NuSMV input format and apply algorithms from Bozzano et al. (2007) to automatically generate Fault Trees. They also compare the efficiency of NuSMV with the sequence generator of AltaRica Data-Flow used in Cecilia OCAS workbench. The sequence generator of AltaRica Data-Flow basically explores all possible combinations of events that lead the system from its nominal state to its failure state. The truncation criterion is the number of events involved in a sequence.

In Griffault et al. (2011), the authors describe algorithms for automatic generation of Fault Trees and critical sequences of events from AltaRica [first version of the language based on constraint automata (Point and Rauzy, 1999; Arnold et al., 2000)] models using symbolic model-checking techniques.

Because of its expressiveness, AltaRica 3.0 models cannot be compiled into Fault Trees just by backward induction on values of flow variables. More elaborated compilation schemes must be applied. Thanks to advanced partitioning techniques we do not need to explore the whole system model in order to discover sequences of events leading the system from its nominal state to its failure state.

7 Conclusions

In this article, we introduced an algorithm for automatic generation of Fault Trees and calculation of minimal cut sets from AltaRica 3.0 models. This algorithm is a

generalisation of the algorithm for AltaRica Data-Flow. AltaRica 3.0 improves AltaRica Data-Flow into two directions: first, its semantics is based on the new underlying mathematical model – GTSs – that makes it possible to represent acausal components and to handle looped systems; second, it provides new constructs to structure models, that greatly improves its capacity of reuse and knowledge capitalisation.

The compilation of AltaRica 3.0 into Fault Trees is of interest for several reasons. Assessment tools for Boolean models are much more efficient than those for states/transitions models. Also, the automated generation of Fault Trees from high level models makes easier their maintenance through the life cycle of systems under study.

However, the price to pay is the loss of sequencing among events: sequences of events are compiled into conjuncts of events. If the GTS is combinatorial, its compilation to Fault Trees is efficient and does not loose information. Many real-life models are relatively simple extensions of reliability block diagrams and, thus, can be compiled efficiently into Fault Trees.

Compilation techniques presented in this article can be applied to other formalisms, beyond AltaRica and GTSs.

Our future works will focus on testing and improvement of the algorithm (and its implementation) for industrial scale models and also on its adaption for the generation of critical sequences of events.

References

Adeline, R., Cardoso, J., Darfeuil, P., Humbert, S. and Seguin, C. (2010) 'Toward a methodology for the AltaRica modelling of multi-physical systems', in Ale, B.J.M., Papazoglou, I.A. and Zio, E. (Eds.): *Proceedings of the European Safety and Reliability Conference, ESREL 2010*, September, ISBN 978-0-415-60427-7, Taylor and Francis Group. Rhodes, Greece.

Arnold, A., Griffault, A., Point, G. and Rauzy, A. (2000) 'The AltaRica language and its semantics', *Fundamenta Informaticae*, Vol. 34, Nos. 2–3, pp.109–124.

Bernard, R., Aubert, J-J., Bieber, P., Merlini, C. and Metge, S. (2007) 'Experiments in model-based safety analysis: flight controls', in Faure, J-M. (Ed.): *Proceedings of the IFAC workshop on Dependable Control of Discrete Systems, DCDS 2007*, June, pp.43–48, ISBN 9781617389948, Curran Associates, Inc., Cachan, France.

Bernard, R., Metge, S., Pouzolz, F., Bieber, P., Griffault, A. and Zeitoun, M. (2008) 'AltaRica refinement for heterogeneous granularity model analysis', *Actes du congrès Lambda – Mu'16*, IMdR (actes électroniques), October, p.2B, Avignon, France.

Bieber, P., Blanquart, J-P., Durrieu, G., Lesens, D., Lucotte, J., Tardy, F., Turin, M., Seguin, C. and Conquet, E. (2008) 'Integration of formal fault analysis in ASSERT: case studies and lessons learnt', *Proceedings of the 4th European Congress Embedded Real Time Software, ERTS 2008*, SIA (electronic proceedings), Code R-2008-01-2B04, January, Toulouse, France.

Boiteau, M., Dutuit, Y., Rauzy, A. and Signoret, J-P. (2006) 'The AltaRica data-flow language in use: assessment of production availability of a multistates system', *Reliability Engineering and System Safety*, Vol. 91, No. 7, pp.747–755.

Bouissou, M. (2005) 'Automated dependability analysis of complex systems with the KB3 workbench: the experience of EDF R&D', *Proceedings of the International Conference on Energy and Environment*.

Bouissou, M., Bouhadana, H., Bannelier, M. and Villatte, N. (1991) 'Knowledge modelling and reliability processing: presentation of the Figaro modelling language and associated tools', *Proceedings of Safecomp'91*.

Bozzano, M., Cimatti, A. and Tapparo, F. (2007) 'Symbolic fault tree analysis for reactive systems', *Proceedings of the 5th International Conference on Automated Technology for Verification and Analysis*, pp.162–176, Springer-Verlag, Berlin, Heidelberg.

Bozzano, M., Cimatti, A., Lisagor, O., Mattarei, C., Mover, S., Roveri, M. and Tonetta, S. (2011) 'Symbolic model checking and safety assessment of AltaRica models', *Proceedings of the 11th InternationalWorkshop on Automated Verification of Critical Systems (AVoCS 2011)*, Vol. 46, Electronic Communications of the EASST.

Brameret, P-A., Rauzy, A. and Roussel, J-M. (2013) 'Preliminary system safety analysis with limited Markov chain generation', *Proceedings of the 4th IFAC Workshop on Dependable Control of Discrete Systems, DCDS'2013*, September, pp.13–18, ISBN: 978-3-902823-49-6, ISSN: 1474-6670, International Federation of Automatic Control, York, Great Britain.

Chaudemar, J-C., Bensana, E., Castel, C. and Seguin, C. (2009) 'AltaRica and event-B models for operational safety analysis: unmanned aerial vehicle case study', *Proceedings of the Workshop on Integration of Model-Based Formal Methods and Tools*, Dusseldorf, Germany, February.

de Kleer, J. (1986) 'An assumption based TMS', *Artificial Intelligence*, March, Vol. 278, No. 2, pp.127–162.

Gössler, G. and Sifakis, J. (2005) 'Composition for component-based modeling', *Science of Computer Programming*, Vol. 55, Nos. 1–3, pp.161–183.

Griffault, A. and Vincent, A. (2004) 'The MEC 5 model-checker', *Proceedings of the 16th International Conference on Computed Aided Verification (CAV 2004), Lectures Notes in Computer Science*, July, Vol. 3114, pp.488–491, Springer Verlag, Boston, MA, USA.

Griffault, A., Point, G., Kuntz, F. and Vincent, A. (2011) *Symbolic Computation of Minimal Cuts for AltaRica Models*, Technical report, LaBRI, Université de Bordeaux.

Hatchuel, A., Weil, B. and La théorie, C-K. (2002) 'Fondements et usages d'une théorie unifiée de la conception', *Actes du Colloque 'Sciences de la conception*', March, Lyon, France.

Hibti, M., Friedlhuber, T. and Rauzy, A. (2012) 'Overview of the open-PSA platform', in Virolainen, R. (Ed.): *Proceedings of the International Joint Conference PSAM'11/ESREL'12*, Helsinki, Finland, June.

Humbert, S., Seguin, C., Castel, C. and Bosc, J-M. (2008) 'Deriving safety software requirements from an AltaRica system model', in Harrison, M.D. and Sujan, M-A. (Eds.): *Proceedings of the 27th International Conference on Computer Safety, Reliability, and Security, SAFECOMP'2008, LNCS*, September, Vol. 5219, pp.320–331, ISBN 978-3-540-87697-7 Springer, Newcastle upon Tyne, England.

Joshi, A., Binns, P. and Vestal, S. (2007) 'Automatic generation of fault trees from AADL models', *Proceedings of the ICSE Workshop on Aerospace Software Engineering*, Minneapolis, USA.

Khuu, M.T. (2008) *Contribution à l'accélération de la simulation stochastique sur des modèles AltaRica Data Flow*, Thèse de doctorat, Université de la Méditerrané, Aix-Marseille II.

Majdara, A. and Wakabayashi, T. (2009) 'Component-based modeling of systems for automated fault tree generation', *Reliability Engineering and System Safety*, Vol. 94, No. 6, pp.1076–1086.

Noble, J., Taivalsaari, A. and Moore, I. (1999) *Prototype-Based Programming: Concepts, Languages and Applications*, Springer-Verlag, Berlin and Heidelberg GmbH & Co. K.

Papadopoulos, Y. and Maruhn, M. (2001) 'Model-based synthesis of fault trees from MATLAB-Simulink models', *Proceedings of the 2001 International Conference on Dependable Systems and Networks, DSN '01*, pp.77–82, IEEE Computer Society, Washington, DC, USA.

Pasquini, A., Papadopoulos, Y. and McDermid, J. (1999) 'Hierarchically performed hazard origin and propagation studies', *Computer Safety, Reliability and Security, LNCS*, Vol. 1698, pp.688–688.

Point, G. and Rauzy, A. (1999) 'AltaRica: constraint automata as a description language', *Journal Européen des Systèmes Automatisés*, Vol. 33, Nos. 8–9, pp.1033–1052.

Prosvirnova, T. and Rauzy, A. (2012) 'Guarded transition systems: a Pivot modeling formalism for safety analysis', in Barbet, J.F. (Ed.): *Actes du Congrès Lambda-Mu 18*, Tours, France, October.

Prosvirnova, T., Batteux, M., Brameret, P-A., Cherfi, A., Friedlhuber, T., Roussel, J-M. and Rauzy, A. (2013) 'The AltaRica 3.0 project for model-based safety assessment', *Proceedings of the 4th IFAC Workshop on Dependable Control of Discrete Systems*, DCDS, September, IFAC, York, Great Britain.

Quayzin, X. and Arbaretier, E. (2009) 'Performance modeling of a surveillance mission', *Proceedings of the Annual Reliability and Maintainability Symposium, RAMS'2009*, January, pp.206–211, ISBN 978-1-4244-2508-2, IEEE, FortWorth, Texas, USA.

Rauzy, A. (2002) 'Mode automata and their compilation into fault trees', *Reliability Engineering and System Safety*, Vol. 78, No. 1, pp.1–12.

Rauzy, A. (2008a) 'BDD for reliability studies', in Misra, K.B. (Ed.): *Handbook of Performability Engineering*, pp.381–396, ISBN 978-1-84800-130-5, Elsevier.

Rauzy, A. (2008b) 'Guarded transition systems: a new states/events formalism for reliability studies', *Journal of Risk and Reliability*, Vol. 222, No. 4, pp.495–505.

Rauzy, A. (2012) 'Anatomy of an efficient fault tree assessment engine', in Virolainen, R. (Ed.): *Proceedings of the International Joint Conference PSAM'11/ESREL'12*, Helsinki, Finland, June.

Rauzy, A. (2013) *AltaRica Data-Flow Language Specification*, Technical report, Ecole Polytechnique, Version 2.3.

Riera, D., Milcent, F., Parisot, J. and Clement, E. (2012) 'Dynamic modeling for dependability and safety evaluation: an advance for the analysis of complex systems', in Barbet, J.F. (Ed.): *Actes du Congrès Lambda-Mu 18*, Tours, France, October.

Sagaspe, L. and Bieber, P. (2007) 'Constraint-based design and allocation of shared avionics resources', *Proceedings of the 26th AIAA-IEEE Digital Avionics Systems Conference*, October, pp.2.A.5-1–2.A.5-10, IEEE, Dallas, Texas, USA.

Sghairi, M., De Bonneval, A., Crouzet, Y., Aubert, Y., Brot, P. and Laarouchi, Y. (2009) 'Distributed and reconfigurable architecture for flight control system', *Proceedings of the 28th Digital Avionics Systems Conference (DASC'09)*, October, pp.6.B.2-1–6.B.2-10, Etats-Unis, Orlando, Florida.

Appendix

Table A1 The semantics of actions

$$S0: \frac{}{\langle skip, \sigma, \tau \rangle \to \tau}$$

$$S1: \frac{\tau(v) = ?,\ \sigma(E) \in dom(s)}{\langle v := E, \sigma, \tau \rangle \to \tau\left[\sigma(E)/v\right]} \qquad S2: \frac{\tau(v) = \sigma(E), \sigma(E) \in dom(v)}{\langle v := E, \sigma, \tau \rangle \to \tau}$$

$$S3: \frac{\sigma(E) = ERROR \text{ or } \sigma(E) \notin dom(v) \text{ or } \tau(v) \neq, \sigma(E) \neq \tau(v)}{\langle v := E, \sigma, \tau \rangle \to ERROR}$$

$$S4: \frac{\sigma(C) = TRUE}{\langle v := \text{if } C \text{ then } I, \sigma, \tau \rangle \to \langle I, \sigma, \tau \rangle} \qquad S5: \frac{\sigma(C) = FALSE}{\langle \text{if } C \text{ then } I, \sigma, \tau \rangle \to \tau}$$

$$S6: \frac{\sigma(C) = ERROR}{\langle \text{if } C \text{ then } I, \sigma, \tau \rangle \to ERROR}$$

$$S7: \frac{\langle I, \sigma, \tau \rangle \to \tau'}{\langle I_1; I_2, \sigma, \tau \rangle \to \langle I_2, \sigma, \tau' \rangle} \qquad S8: \frac{\langle I_2, \sigma, \tau \rangle \to \tau'}{\langle I_1; I_2, \sigma, \tau \rangle \to \langle I_1, \sigma, \tau' \rangle}$$

$$S9: \frac{\langle I_1, \sigma, \tau \rangle \to \langle I_1', \sigma, \tau' \rangle}{\langle I_1; I_2, \sigma, \tau \rangle \to \langle I_1'; I_2, \sigma, \tau' \rangle} \qquad S10: \frac{\langle I_2, \sigma, \tau \rangle \to \langle I_2', \sigma, \tau' \rangle}{\langle I_1; I_2, \sigma, \tau \rangle \to \langle I_1; I_2', \sigma, \tau' \rangle}$$

$$S11: \frac{\langle I_1, \sigma, \tau \rangle \to ERROR}{\langle I_1; I_2, \sigma, \tau \rangle \to ERROR} \qquad S12: \frac{\langle I_2, \sigma, \tau \rangle \to ERROR}{\langle I_1; I_2, \sigma, \tau \rangle \to ERROR}$$

Table A2 The semantics of assertions

$$S0: \frac{}{\langle skip, \tau \rangle \to \tau}$$

$$S1: \frac{\tau(v) = ?,\ \tau(E) \neq ?, \tau(E) \in dom(v)}{\langle v := E, \tau \rangle \to \tau\left[\tau(E)/v\right]} \qquad S2: \frac{\tau(v) = \tau(E), \tau(E) \in dom(v)}{\langle v := E, \tau \rangle \to \tau}$$

$$S3: \frac{\tau(E) = ERROR \text{ or } \tau(E) \notin dom(v) \text{ or } \tau(v) \neq ?, \tau(E) \neq \tau(v)}{\langle v := E, \tau \rangle \to ERROR}$$

$$S4: \frac{\tau(C) = TRUE}{\langle v := \text{if } C \text{ then } I, \tau \rangle \to \langle I, \tau \rangle} \qquad S5: \frac{\tau(C) = FALSE}{\langle \text{if } C \text{ then } I, \tau \rangle \to \tau}$$

$$S6: \frac{\tau(C) = ERROR}{\langle \text{if } C \text{ then } I, \tau \rangle \to ERROR}$$

$$S7: \frac{\langle I_1, \tau \rangle \to \tau'}{\langle I_1; I_2, \tau \rangle \to \langle I_2, \tau' \rangle} \qquad S8: \frac{\langle I_2, \tau \rangle \to \tau'}{\langle I_1; I_2, \tau \rangle \to \langle I_1; I_2', \tau' \rangle}$$

$$S9: \frac{\langle I_1, \tau \rangle \to \langle I_1', \tau' \rangle}{\langle I_1; I_2, \tau \rangle \to \langle I_1'; I_2, \tau' \rangle} \qquad S10: \frac{\langle I_2, \tau \rangle \to \langle I_2', \tau' \rangle}{\langle I_1; I_2, \tau \rangle \to \langle I_1; I_2', \tau' \rangle}$$

$$S11: \frac{\langle I_1, \tau \rangle \to ERROR}{\langle I_1; I_2, \tau \rangle \to ERROR} \qquad S12: \frac{\langle I_2, \tau \rangle \to ERROR}{\langle I_1; I_2, \tau \rangle \to ERROR}$$