Symbolic computation of minimal cuts for AltaRica models

Alain Griffault, Gérald Point, Fabien Kuntz, Aymeric Vincent

LaBRI

Symbolic computation of minimal cuts for AltaRica models

Research Report RR-1456-11

Alain Griffault[1], Gérald Point[1], Fabien Kuntz[1,2] and Aymeric Vincent[1]

{firtname.lastname}@labri.fr

September 30, 2011

1 - LaBRI, Université de Bordeaux, 33405 Talence cedex

2 - Thales Avionics S.A., F-31036 Toulouse, France

Abstract

AltaRica[1, 13] tools developped at LaBRI have always been dedicated to model-checking thus focusing analysis onto functional aspects of systems. In this report we are interested by a problem encountered in safety assessment or diagnosis domains: the computation of all failure scenarios. This problem consists to determine preponderant sequences of failures of elementary components that lead the system into a critical state. While model-checkers usually look for a counter-example of a good property of the system, here we want to compute *all* the most significant paths to a bad state.

The solution presented in this paper is mainly a mix of existing works taken from the literature[17, 5, 16, 13]; however, in order to be able to treat large models, we have also implemented a preprocessing algorithm that permits to simplify the input model w.r.t. to specified unwanted states.

This work has been realized under the grant of Thales Avionics (Toulouse).

Contents

Chapter 1

Introduction

In this report we are interested by the computation of failures scenarios for AltaRica models. Such computations interest several communities that try to make safe critical systems. However, each community studies these critical sequences from a different point of view.

In the formal verification community one looks for what is, more or less, a bug of the system. Engineers specify sets of functional requirements that must be fulfilled by the system and verification tools check if behaviours described by the model satisfy these properties. If this is not the case then, a failure scenario is returned by the tool as a couter-example.

From the point of view of safety assessment community the system will eventually fail. Engineers specify (global) failures of the system or at least of its critical functions and they want to determine and to minimize the probability of this failure. For this aim they have to identify weakest parts of the system. In this context it is clear that only one failure scenario is not sufficient and that all sequences of elementary failures (i.e., those of basic components) must be considered.

In the domain of diagnosis, behaviours are studied afterwards: we observe an abnormal (not necessarily critical) behaviour of the system and we want to understand what is happened to fix the problem. Here we have the same point of view than safety assessment except that the failure can be a more local phenomenon. Yet, the whole set of behaviours that lead the system in observed state have to be investigated.

In this study we adopt the point of view of safety assessment and diagnosis community. We want to compute the whole set of scenarios that produces a failure or a deviation of the nominal behaviour of the system.

The remain of this introducing chapter formalizes the concept of scenario and presents what will be computed by algorithms described in next chapters.

1.1 What is a failure scenario ?

Each community has its own point of view of what can be a *failure scenario* and these points of view obviously impact the final object that algorithms have to produce. For instance, the counter-example generated by a model-checker is essentially the sequence of states and events that lead the model into a faulty state; actually, this is the minimum data that we expect in order to fix the bug.

In safety assessment domain, we want to lower the probability of a critical event. This probability is evaluated with respect to failure rate of elementary components of the system. In this domain, scenarios are essentially composed by failure events of components. Actual failure scenarios can contain occurrences of non-failure events (e.g. repair, reconfigurations, ...) but, since they are deterministic event (from probabilistic point of view), they are ignored.

When we make a diagnosis of an abnormal behaviour of the system one want to identify what are the components that have contributed to the deviation from the nominal behaviour. Maintenance teams need a "catalog" of possible combinations of components that have to be checked and possibly replaced. Yet this domain is mainly interested by elementary failures of components.

These remarks justify that scenarios could be abstracted in two ways. Firstly, certain events are distinguished from others (these are for instance elementary failures) and only these events should appear in results produced by algorithms. In the sequel we will say that such interesting events are *visible*. This notion of visibility is different from *observability* encountered in classical frameworks on diagnosis[19] or *controlability*[15]. Besides, in these frameworks, failure events are considered unobservable and uncontrollable. In our context, visibility attribute only identifies events that must be kept in the result.

The second abstraction is related to the relevance of knowing order of appearance of events. Often, failures are independant events and any interleaving of elementary events should lead the system in a same state. Obviously this property is not always satisfied. However ignoring the logical ordering of events has the advantage of decreasing the complexity of the result while keeping pessimistic. In addition, from a maintenance point of view, we are only interested by the identification of faulty components. This is the reason why we compute two kind of scenarios:

1. List of events, called *words* or *sequences* in the sequel, that explicit the order of occurrence of events. A same set of events may occur under different ordering.

2. Sets of elementary events called *cuts* by safety assessment community. Under this form the order of appearance is forgotten.

1.2 Failure scenarios of an AltaRica model

An AltaRica model is a hierarchy of abstract machines called *constraint automata*[14]. AltaRica semantics[13] permits to flatten a model into a single constraint automaton equivalent to the whole hierarchical model. The semantics of this flat constraint automaton captures all behaviours of the described system. This semantics is formalized by the mean of a *interfaced labelled transition system*[1]. In the context of this study, since the concept of interface is not relevant, we will restrict us to *labelled transition systems* (LTS). The formal definition of constraint automata is postponed in chapter 5 but their semantics in terms of LTS is straightforward.

Definition 1 (Labelled Transition Systems) *A* labelled transition system *(LTS) is a tuple* $\mathcal{A} = \langle S, I, \Sigma, T \rangle$ *where:*

- S *is a finite set of states.*
- $I \subseteq S$ *is the set of initial states.*
- Σ *is a finite alphabet.*
- $T \subseteq (S \times \Sigma \times S)$ *is a set of transitions.*

A labelled transition system is essentially a graph whose vertices are states of the system and edges between two states are labelled by an event of the model. We denote by $s_1 \xrightarrow{e} s_2$ the fact that $(s_1, e, s_2) \in T$.

A *path* is a sequence $p = s_0, e_1, s_1, \ldots, e_n, s_n$ where each s_i is a state in S and e_i is a letter in Σ and such that for each $i = 1 \ldots n$, $s_{i-1} \xrightarrow{e_i} s_i$. n is the length of the path. We denote s_0 by $\alpha(p)$ and s_n by $\beta(p)$. The word $\lambda(p) = e_1 \ldots e_n \in \Sigma^*$ is called the *trace* of p. If the length of p is 0 then its trace is the empty word: $\lambda(p) = \epsilon$. We denote by $|w|$ the length of a word w.

A *run* is a path r starting from an initial state of $\mathcal{A}$, i.e., such that $\alpha(r) \in I$. For any LTS $\mathcal{A}$ its set of runs is denoted $Run(\mathcal{A})$.

In this study we are interested by traces of runs that leads the system into critical states. Given a LTS $\mathcal{A} = \langle \mathcal{S}, I, \Sigma, T \rangle$ and a set of critical states $F \subseteq S$ we define the set of traces leading into F by:

$$L(\mathcal{A}, F) = \{\lambda(r) \mid r \in Run(\mathcal{A}) \wedge \beta(r) \in F\}$$

$L(\mathcal{A}, F)$ contains all the scenarios that produce the failure of the modelled system. Elements of $L(\mathcal{A}, F)$ are event of any nature: repairs, failures, (re)configurations, tests, ... As said at the beginning of this chapter, in the context of safety assessment, as well as for diagnosis purpose, not all kinds of event are interesting; in general, we want to retain only failure events. Let $V \subseteq \Sigma$ be a set of so-called *visible* events. We define the erasing morphism $\Phi_V : \Sigma^\star \rightarrow V^\star$:

- $\Phi_V(\epsilon) = \epsilon$
- $\Phi_V(a) = \epsilon$ if $a \notin V$

- $\Phi_V(a) = a$ if $a \in V$
- $\Phi_V(w_1.w_2) = \Phi_V(w_1).\Phi_V(w_2)$ for any w_1, w_2 in $\Sigma^\star$

Using Φ_V we can formalize the set of failure sequences as:

$$Sequences(\mathcal{A}, F, V) = \{\Phi_V(w) \mid w \in L(\mathcal{A}, F)\}$$

Computing this set could permit to study chains of events that produce specified failures of the system. However $Sequences(\mathcal{A}, F, V)$ is an huge set and can even be infinite if there exist repairable components. Since this set is a rational language it can be stored in an finite automaton recognizing words. But in this case it remains the issue of exploiting this automaton regarding safety assessment or diagnosis which is not an obvious task. In the context of this study we restrict us to sequences of length bounded by some integer k specified by the user:

$$Sequences(\mathcal{A}, F, V, k) = \{w \mid w \in Sequences(\mathcal{A}, F, V) \wedge |w| \leq k\}$$

As explained in introduction, engineers prefer, at least at first, to forget the ordering of events and to retain only what are the components that failed during scenarios; these sets of components or elementary failures are called *cuts*. We associate to any sequence of events w, its corresponding set of events $cut(w) : \Sigma^\star \to 2^\Sigma$ defined by:

- $cut(\epsilon) = \emptyset$
- $cut(a) = \{a\}$ for any $a \in \Sigma$
- $cut(w_1.w_2) = cut(w_1) \cup cut(w_2)$ for any w_1, w_2 in $\Sigma^\star$

and we have to produce the set $Cuts(\mathcal{A}, F, V)$ defined by:

$$Cuts(\mathcal{A}, F, V) = \{cut(w) \mid w \in Sequences(\mathcal{A}, F, V)\}$$

Even if $Sequences(\mathcal{A}, F, V, k)$ and $Cuts(\mathcal{A}, F, V)$ are strong abstraction of what actually produces the global failure of the system or an observed deviation from its nomimal mode, these sets can contain redundant data that must be removed. This is especially the case of coherent systems where adding new elementary failures once F has been reached, can not restore the nominal mode of the system. As a consequence, it is generally the case that sequences of $Sequences(\mathcal{A}, F, V, k)$ or sets in $Cuts(\mathcal{A}, F, V)$ can be augmented with a new event in V while remaining a failure scenario.

To handle this problem, diagnosis community[18, 6] applies the *parsimony principle* and retain only preponderant scenarios considering that those oneswith redundant data should be helpless to fix the failure.

Applying this principle to $Sequences(\mathcal{A}, F, V, k)$ and $Cuts(\mathcal{A}, F, V)$ requires the definition of two criteria that express that a sequence or a cut is more interesting than another one. These criteria are subword order for sequences (denoted $\sqsubseteq$) and inclusion for cuts ($\subseteq$). Our goal in this study is thus to compute minimal elements of computed sets of scenarios that is:

$$MinSequences(\mathcal{A}, F, V, k) = min_{\sqsubseteq}\{w \mid w \in Sequences(\mathcal{A}, F, V, k)\}$$

$$MinCuts(\mathcal{A}, F, V, k) = min_{\subseteq}\{w \mid w \in Cuts(\mathcal{A}, F, V)\}$$

1.3 Organization of the paper

The following chapter describe algorithms that have been implemented into ARC to compute $MinSequences$ and $MinCuts$ for an AltaRica model. In the next chapter we present observer based algorithms that permit to compute sets of cuts and sets of bounded sequences. The third chapter describes a data structure called RLDD used to store finite sequences and to minimize sets of sequences according to subword order ($\sqsubseteq$). The fourth chapter explains how to obtain $MinSequences$ and $MinCuts$ from $Sequences$ and $Cuts$. The fifth chapter shows how large models can be preprocessed in order to be treated by ARC. We have tested our algorithms on several models; results are presented in the sixth chapter.

Chapter 2

Observer based algorithms to compute scenarios

In this chapter we present algorithms that compute sets of cuts or sets of sequences for a given targeted set of configurations specified by a Boolean formula ϕ. Theses algorithms consist essentially in the use of an observer that records occurrences of visible events; the observation is then used to synthesize a Boolean formula that models the set of cuts (or sequences) that generate configurations satisfying ϕ.

2.1 Principle of an observer

The usual way to analyze a system is to model it and then use some algorithm to get informations about modeled behaviours. Even with efficient algorithms this direct method encounters well-known problems that are memory exhaustion and excessive time consumption. These problems occur because we usually study the whole state-graph of the model regardless of the studied property.

One way to overcome these issues is to design new algorithms that make their best effort to build only a "small" part of the whole state-graph (e.g. on-the-fly algorithms[20] or compositional model-checking[4]). Another way is to apply existing algorithm to a model with less or restricted behaviours. Since engineers can not produce a new model for each studied property, the best methods is to extend the unique model of the system with additional data (w.r.t to the studied property) that should reduce the number of possible behaviours.

An *observer* is a new component that is added and synchronized with the model of the system. The aim of this component is twofold:

- First, it records data related to behaviours of the model. The observer can store values, remember that some events occur and so on.

- Second, according to data it records, the observer can inhibit some behaviours of the system; those considered irrelevant for the computed property.

Of course this method strongly depends on the ability of the underlying formalism to allow a component to observe and constraint the rest of the model. A mean to get round this difficulty is to create the observer in the analysis tool, once the model has been compiled into low-level data-structures. The algorithms proposed in this chapter are based on this approach.

2.2 Visible events

Our objective is to generate cuts or sequences of events that yield unexpected configurations. To realize that, we observe sequences of global events produced by the semantics of the model. These global events are vectors of elementary events; thus, we observe sequences of vectors. Among these vectors just some of them are kept for the result; these selected events are said *visible*. In this setting, two choices had to be made: How visible global events are specified ? And, what information is pertinent, vectors or elementary events that compose them ? The first point influence mainly methodology to increase efficiency of algorithms. The second point influence the design of the algorithm itself and returned results.

2.2.1 How visible global events are specified ?

Since observed events are often elementary events like failures, the simplest and the most natural way to specify visible events is by attaching an information onto declared events of nodes. One could have define a similar mechanism for synchronization vectors but, since AltaRica semantics induces implicit (and sometimes complex) constructions of synchronization vectors, it would have been confusing or misleading for users.

Our choice has two advantages. First the language already has syntax to attach *tags* (also called *attributes*) to events without any impact on the semantics of the model. Second it permits to identify what are interesting elementary events in a global event; this can be used to refine displayed results.

In practice *tags* follow the declaration of the event:

```
event ev1, ev2 : tag1, ..., tagn;
```

Tags are attached to the list of events that immediately precedes it. In the following example, `stuck` and `broken` are tagged with the attribute `failures` while no tag is attached to `action` and `repair` events.

```
node Component
  event
    action, repair;
    stuck, broken : failures;
  ..
edon
```

Global events inherit all attributes of elementary events that compose them.

Example 1 *The following example models a component that is started when a second one fails. To model this phenomenon, the start-up of the first one is weakly synchronized with the failure of the second one.*

```
node Component1
  event start: not_a_failure;
edon

node Component2
  event stuck : failure;
edon

node Main
  sub c1 : Component1;
      c2 : Component1
  sync <c1.start?, c2.stuck>;
edon
```

The global events generated by the synchronization of events are:

- $\langle \epsilon,$ `c1.start,` `c2.broken`$\rangle$ *tagged with attributes* `not_a_failure` *and* `failure`*;*
- $\langle \epsilon,$ `c1.`$\epsilon,$ `c2.broken`$\rangle$ *tagged with only the attribute* `failure`*;*

When used within ARC, any attribute can be used to specify visible events. Note that ARC also allows to specified disabled events that are ignored while the computations of cuts or sequences. Disabling attributes are applied to vectors using the same rules that for visible events. In the case where a vector possesses both a disabled event and a visible one; disabling prevails.

2.2.2 Global or elementary events ?

Choosing the kind of data that must be kept in the result depends on the expected abstraction level. When computing cuts, we adopt a coarse point of view on what produces unexpected configurations. Each minimal cut identifies components that contribute to the failure of the system. Clearly, in the case of computation of cuts, elementary events are the important data.

When sequences are computed, we look for detailed information on the logical ordering of events that yield the failure and even the fact that a different ordering does not produce the unexpected configuration becomes a pertinent information. For this kind of study it makes no sense to loose the fact that some elementary events have occurred simulateously; especially if one wants to simulate failure scenarios using some tool. Clearly, in case of sequences computation global events should be returned by algorithms.

The choice between elementary events and vectors has an important impact on the result after minimization. Since basic objects that compose computed scenarios are not the same, the set of minimal cuts and the set of minimal sequences for a same model can be completely different.

To illustrate this last point, consider the following models that represent two components that can fail. The failure of the second one induces the failure of the first one. Thus, we declare two synchronization vectors: one where the first component fails alone and one where both components fail simultaneously.

```
node Component
  event failure : vis;
  state mode : { ok, nok }; init mode := ok;
  trans mode = ok |- failure -> mode := nok;
edon

node Main
  sub
    c1, c2 : Component;
  sync
    <c1.failure>;
    <c1.failure, c2.failure>;
edon
```

Now suppose we are interested by the unexpected configuration where `c1` has failed. In the case of minimal cuts we get only one scenario:

```
(c1.failure)
```

while in the case of sequences we get two:

```
(<c1.failure, c2.failure>)
(c1.failure)
```

2.3 An observer to compute set of cuts

In [17], A. Rauzy proposes an algorithm to compute cuts from the explicit state-graph representing the semantics of an AltaRica model. The principle of this algorithm is to compute couples $(c, \vec{e})$ where c is a configuration and $\vec{e}$ is a vector of bits (one bit $\vec{e}[e_i]$ per visible event e_i) such that:

1. There exists a path p, from an initial configuration to c;
2. p is labelled by events e_i such that $\vec{e}[e_i] = 1$.

Each vector $\vec{e}$ is then translated into the Boolean formula $F_{\vec{e}}$:

$$F_{\vec{e}} \stackrel{\text{def}}{=} \bigwedge_{\vec{e}[e_i]=1} e_i \wedge \bigwedge_{\vec{e}[e_i]=0} \neg e_i$$

Finally the formula F_ϕ representing the cuts for a specification of unwanted states ϕ is the disjunction of formulae $F_{\vec{e}}$ for which there exists a configuration that satisfy ϕ:

$$F_\phi \stackrel{\text{def}}{=} \bigvee_{\{(c,\vec{e})|c\models\phi\}} F_{\vec{e}}$$

Algorithm 1 Computation of vectors encoding cuts

1: $C \leftarrow \{\langle c, \vec{0}\rangle | c \in Initial\}$, $D \leftarrow \emptyset$
2: **while** $C \neq \emptyset$ **do**
3: let $\langle c, \vec{e}\rangle \in C$
4: $C \leftarrow C \setminus \{\langle c, \vec{e}\rangle\}$, $D \leftarrow D \cup \{\langle c, \vec{e}\rangle\}$
5: **for all** transitions $t = \langle c, e_i, c'\rangle$ **do**
6: $C \leftarrow C \cup \{\langle c', \vec{e}[e_i] \leftarrow 1\rangle\}$
7: **end for**
8: **end while**

The algorithm 2.3 is the one proposed by A. Rauzy in [17] (the author did not give more about data structure used by the algorithm); we omit details related to *mode automata*.

Rauzy's algorithm is essentially the computation of reachable configurations augmented with $\vec{e}$ vectors. One way to implement this algorithm is to directly embed vectors into the model by the mean of additional Boolean variables. Actually, the model is augmented with an observer that records occurrences of visible events into Boolean variables. The augmentation consists in the following steps:

1. Add to the constraint automaton a new Boolean variable v_e for each visible event e. Each variable v_e is initialized with `false`.

2. For each macro-transition labelled by a visible event e, add the assignment $v_e \leftarrow$ `true`.

This modification of the model is illustrated on the figure 2.1.

The above modification of the model does not create deadlock nor change sequences of events (actually semantics are bisimilar); however state graphs can be non-isomorphic as show on figure 2.2.

2.4 Observation of sequences

There exist important differences in the nature of the expected result when computing cuts or sequences. These differences make the problem of sequences generation a bit harder because more informations have to be recorded.

- First we have to remember the logical ordering of events that lead the system in an unexpected configuration.

- Second, events may occur several times. Cuts forget multiple occurrences of a same event and thus guarantee that what is computed is a possibly huge but finite set. In the case of sequences, the result can contain an infinite number of elements. Since models have a finite number of configurations, if an infinite number of sequences yield a same unexpected

```
node Generator
   flow
     out : bool;
   state
     mode : { ok, ko };
     on : bool;
   init
     mode := ok, on := false;
   event
     push;
     fails : visible;
   assert
     if mode = ko then out = false
          else out = true;
   trans
      mode = ok |- push -> on := not
            on;
      mode = ok |- fails -> mode :=
           ko;
edon
```

```
node ObservedGenerator
   flow
     out : bool;
   state
     mode : { ok, ko };
     on : bool;
   init
     mode := ok, on := false;
   state
     fails_event : bool;
   init
     fails_event := false;
   event
     push, fails;
   assert
     if mode = ko then out = false
          else out = true;
   trans
      mode = ok |- push -> on := not
            on;
      mode = ok |- fails -> mode :=
           ko, fails_event := true;
edon
```

Figure 2.1: An AltaRica node (left) and the observed version of the same node (right). One wants to observe occurrences of the failure event called `fails`. A new Boolean variable `fails_event` is added to the model and transitions labelled by `fails` now set to `true` the new variable.

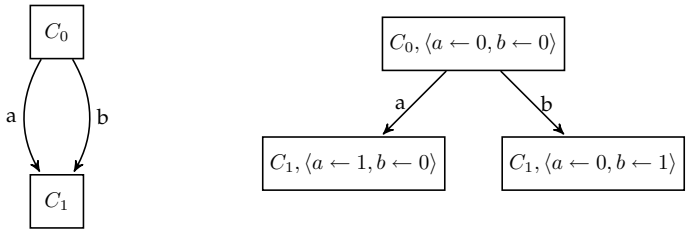


Figure 2.2: Even if configurations are reachable in both graphs, the state graph of the observed model is impacted by vectors that records occurrences of events.

state then there exists a sub-sequence that can be repeated as many time as we want (i.e a loop). Clearly such sequences are irrelevant for risk assessment or diagnosis and algorithms should ignore them.

Our objective in this section is to design an algorithm similar to the one used to compute cuts. This latter consists to synchronize the model with an observer that records sufficient data to encode scenarios. Since AltaRica does not support dynamic creation of variables, an upper bound to the state space required to store the current sequence must be known in advance. But, contrary to cuts, it is difficult to get this bound suitable to store the longest sequence (remember that events can be repeated). These considerations leads us to design the algorithm with the hypothesis that the length of sequences, the number of visible events, is fixed in advance by the user. Under this hypothesis it becomes feasible to write an observer that requires only a "small" amount of memory cells (i.e. variable) to encode sequences. Actually, the number of additional variables should remain reasonable because, even if scenarios composed of more than five events are theoretically possible, they are, from a probabilistic point of view, extremely rare; thus users should look for sequence of length less or equal to 4 visible events.

2.4.1 Description of the algorithm

The algorithm proposed here is parameterized by the set of visible events E_V and the maximal number k of visible events that can occur in a sequence. As already said, we follow the same principles than for algorithm used to compute cuts but this time we have to take into account the ordering of events and to translate the resulting DD into a set of sequences.

To record events and their logical order of appearance we can use at least two solutions reviewed in following subsections.

2.4.2 Memorizing *when* events occurs

First we can memorize *when* an event occurs. That means we have to store the logical instants when each visible event occurs. To realize this observer we use integer variables seen as vectors of bits. Each bit models a "visible" instant[1]. This solution requires two operations:

- Firstly, we create a global integer variable $tick$ that ranges in $[1, 2^{k-1}]$ used to model the tick of visible events; k is the maximal length specified by the user. After each occurrence of a visible event the variable is multiplied by 2 i.e the bit encoding the instant is shifted by one position. $tick$ is initialized to 1.

[1]This means that, using this implementation, sequences must be bounded by the number of bits of a computer word (e.g. 32 or 64 bits).

- Then, for each visible event $e \in E_V$ we create an integer variable v_e that takes its values into the range $[0, 2^k - 1]$ and is initialized with 0. To remember that e occurs at the current instant we add $tick$ to the current value of v_e. For instance if $abac$ is a sequence of visible events then, reached configurations should be such that $v_a = 2^0 + 2^2 = 5$, $v_b = 2^1 = 2$ and $v_c = 2^3 = 8$.

Figure 2.3 describes a small system simply composed of two parallel components that may fail. Figure 2.4 gives the constraint automaton of this system decorated with variables as presented above. If for this system we want to obtain sequences that yield configurations such that `c[0].s = nok` within 3 visible events, then we get a DD that encodes the relation, given table 2.1 (page 18), over the three variables of the observer (the fourth column indicates the corresponding sequence – `c[`i`]` is used for `c[`i`].failure`):

```
node Component
  event
    action, repair;
    failure :  visible;
  state
    s :  { ok, nok };
  init
    s := ok;
  trans
    s = ok |- failure -> s := nok;
    s = nok |- repair -> s := ok;
    s = ok |- action -> ;
edon
```

```
node Main
  sub
    c :  Component[2];
edon
```

Figure 2.3: A simple model composed of two parallel components that may fail.

Memorizing *what* is the i^{th} event

The second method consists to memorize *what* happens at each instant. In other terms we have to remember the fact that the i^{th} (visible) event was some visible event e. To implement this solution the observer needs only k variables:

- First we assign to each event $e \in E_V$ an integer $id(e)$ in the range $[1, N]$ where $N = |E_V|$ is the number of visible events.
- Then, we add k variables $p_1, \ldots, p_k$, taking their values in the range $[0, N]$ and initialized with 0. Each p_i stores the event that occurs at the i^{th} instant. p_i is 0 if there is no i^{th} event. Note that we have to ensure (see below) that if p_i equals 0 then for all $j > i$, p_j also equals 0.
- Finally, for each macro-transition labelled with a visible event $e \in E_V$:

```
node DecoratedMain
  state
    'c[0].s' :  { ok, nok };
    'c[1].s' :  { ok, nok };
    tick :  [1,4];
    'v_c[0].failure' :  [0,7];
    'v_c[1].failure' :  [0,7];
  init
    'c[1].s' := ok, 'c[0].s' := ok,
    tick := 1, 'v_c[0].failure' := 0, 'v_c[1].failure' := 0;
  event
    'c[0].failure', 'c[1].failure' :  visible;
    'c[0].repair', 'c[1].repair';
  trans
    ('c[1].s' = nok) |- 'c[1].repair'
          -> 'c[1].s' := ok;
    ('c[1].s' = ok) |- 'c[1].failure'
          -> 'c[1].s' := nok,
          tick := 2 * tick,
          'v_c[1].failure' := 'v_c[1].failure' + tick;
    ('c[0].s' = nok) |- 'c[0].repair'
          -> 'c[0].s' := ok;
    ('c[0].s' = ok) |- 'c[0].failure'
          -> 'c[0].s' := nok,
          tick := 2 * tick,
          'v_c[0].failure' := 'v_c[0].failure' + tick;
edon
```

Figure 2.4: The constraint automaton is obtained by flattening the model given on figure 2.3; then it is decorated to observe up to $k = 3$ occurrences of events tagged with the `visible` attribute i.e $E_V = \{$`c[0].failure`, `c[1].failure`$\}$. Additional data inserted to model the observer are underlined.

tick	'v_c[0].failure'	'v_c[1].failure'	sequence
1	1	0	c[0]
2	1	2	c[0] c[1]
2	2	1	c[1] c[0]
2	3	0	c[0] c[0]
4	1	6	c[0] c[1] c[1]
4	2	5	c[1] c[0] c[1]
4	3	4	c[0] c[0] c[1]
4	4	3	c[1] c[1] c[0]
4	5	2	c[0] c[1] c[0]
4	6	1	c[1] c[0] c[0]
4	7	0	c[0] c[0] c[0]

Table 2.1: Sequences that yield `c[0].s = nok`. The first column gives the last value of $tick$ variable. The next two columns give values of variables that store occurrences of failures when the unexpected configuration is reached and the last column gives the corresponding sequence of failures. Grey line for instance indicates that the sequence contains 3 events ($tick = 2^{3-1} = 4$). The failure of $c[0]$ occurs at the second position because `'v_c[0].failure'` is equal to 2 i.e the bitvector 10. The failure of $c[1]$ occurs at the first and third visible instant because `'v_c[1].failure'` is equal to 5 i.e the bitvector 101.

- We extend the guard with the condition $p_k = 0$. This means that after the k^{th} visible event, all visible events are disabled[2].
- We add assignments:
  - $p_1 :=$ `if` $p_1 = 0$ `then` $id(e)$ `else` p_1
  - $p_2 :=$ `if` $p_1 \neq 0 \wedge p_2 = 0$ `then` $id(e)$ `else` p_2
  - ...
  - $p_k :=$ `if` $p_1 \neq 0 \wedge \cdots \wedge p_{k-1} \neq 0 \wedge p_k = 0$ `then` $id(e)$ `else` p_k

  which mean that each p_i receives the event encoded by $id(e)$ if and only if for all visible instant $j < i$, $p_j \neq 0$ i.e visible events occur before instant i and no event has been recorded at instant i.

Figure 2.5 (page 19) applies this method on the same system than for the previous example 2.4. For this observer we obtain the relation, given table 2.2 (page 20), over p_is variables ($c[0].failure$ is assigned value 1 and $c[1].failure$ the value 2).

[2] A similar disabling condition also exists for the first methods but it is implicit. Actually, since $tick$ is strictly increasing, it can not be assigned after the k^{th} visible event

```
node DecoratedMain
  state
    'c[0].s' :  { ok, nok };
    'c[1].s' :  { ok, nok };
    p_1 :  [0,2];
    p_2 :  [0,2];
    p_3 :  [0,2];
  init
    'c[1].s' := ok, 'c[0].s' := ok, p_1 := 0, p_2 := 0, p_3 := 0;
  event
    'c[0].failure', 'c[1].failure' :  visible;
    'c[0].repair', 'c[1].repair';
  trans
    ('c[1].s' = nok) |- 'c[1].repair' -> 'c[1].s' := ok;
    ('c[1].s'= ok) & (p_1 = 0 | p_2 = 0 | p_3 = 0)
             |-'c[1].failure' ->
               'c[1].s' := nok,
               p_1 := if p_1 = 0 then 2 else p_1,
               p_2 := if p_1 != 0 & p_2 = 0 then 2 else p_2,
               p_3 := if p_1 != 0 & p_2 != 0 & p_3 = 0 then 2 else p_3;

    ('c[0].s' = nok) |- 'c[0].repair' -> 'c[0].s' := ok;
    ('c[0].s' = ok) & (p_1 = 0 | p_2 = 0 | p_3 = 0)
             |- 'c[0].failure' -> 'c[0].s' := nok,
               p_1 := if p_1 = 0 then 1 else p_1,
               p_2 := if p_1 != 0 & p_2 = 0 then 1 else p_2,
               p_3 := if p_1 != 0 & p_2 != 0 & p_3 = 0 then 1 else p_3;
edon
```

Figure 2.5: Augmented constraint automaton for the same system and diagnosis objective than previously for example on figure 2.3. This time, automaton is decorated using variables (p_is) that memorize what happens at the i^{th} visible tick.

p_1	p_2	p_3	sequence
1	0	0	c[0]
1	1	0	c[0] c[0]
1	2	0	c[0] c[1]
2	1	0	c[1] c[0]
1	1	1	c[0] c[0] c[0]
1	1	2	c[0] c[0] c[1]
1	2	1	c[0] c[1] c[0]
1	2	2	c[0] c[1] c[1]
2	1	1	c[1] c[0] c[0]
2	1	2	c[1] c[0] c[1]
2	2	1	c[1] c[1] c[0]

Table 2.2: Sequences that yield `c[0].s = nok` computed using the second encoding. The first three columns give values of p_i variables i.e. which visible event has occurred at the i^{th} position. Last column gives the corresponding sequence. Value 0 means *no event occurs*, 1 means failure of $c[0]$ and 2 means failure of $c[1]$. Grey line for instance indicates that the sequence contains 2 events because the last non-zero p_i variable is p_2. The first event is the failure of $c[1]$ (value is 2) and the second event is the failure of $c[0]$ (value is 1).

Comparison of methods

Both methods have to be used within reachability analysis based on decision diagrams. Even if observer partially reduces behaviours of the system (occurrences of failures are stopped after the k^{th} one), the introduction of new variables should not reduce the size of computed relations, especially for intermediate ones that are known to be larger than the final relation (the one encoding reachable configurations).

Clearly the first method has the drawback to generate lots of variables. If we consider the largest model treated during our experiments (cf. chapter chapter 6), reduced automata can have around 20 and 80 events. The second method produces only k variables but each variable takes as many values as there exists visible events and thus the size of computed relations is unpredictable. However decision diagrams implemented in ARC use a compact encoding of ranges that should help to keep the size of relations manageable.

Although the second method seems to be the best one it has a second minor drawback. Up to now, we have considered events of the flattened constraint automaton; that means the observer memorizes vectors of events and not elementary events (those specified in `nodes`). It happens that for models (see 6), elementary events are not explicitly synchronized so there is a one-to-one correspondence between elementary and global events. The first method is able to handle both kind of events; it suffices to create a variable for each elementary event and to assign in transitions each elementary events appearing in the

labelling vector. The second method can handle only global events. It could be adapted by duplicating p_is variables as many time as there is events in the largest vector but the method would become too costly. In fact the handling of vectors can be postponed, if actually necessary, to the end of the computation process.

2.5 Implementation

ARC implements the algorithm that computes cuts and the second algorithm (section 2.4.2) for sequences. The principle of these implementations is quite simple; it is based on the symbolic computation of reachable configurations already implemented in ARC. We illustrate this algorithm on figure 2.6.

1. After the computation of the flat semantics[13], variables of the observer are added to the constraint automaton equivalent to the original model.
2. ARC symbolically computes the set of reachable configurations of the observed model $\mathcal{A} \times Obs$. This set is represented by a Decision Diagram (DD)[5]. This DD is represented in grey on the figure. For the sake of clarity we have assumed that variables of the observer are at the bottom of the diagram but in practice all variables are mixed.
3. Then, the set of reachable configuration is intersected with those that satisfy ϕ. This gives us a new DD depicted in red and green.
4. This second DD is then projected on the observer part i.e., *event* variables v_e's for cuts or p_is variables for sequences. This is the green DD of the figure.
5. This last DD contains all data necessary to produce cuts or sequences; it suffices to translate it into the appropriate format to get the result: either Boolean formulas or list of sequences.

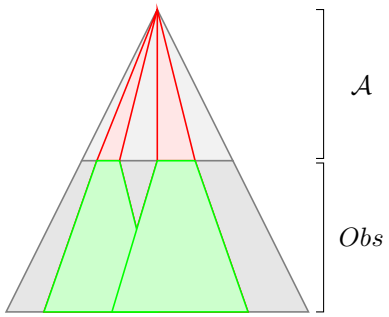


Figure 2.6: DDs computed by the observer-based algorithms.

Remark about sequences: ARC produces sequences using the second algorithm (section 2.4.2). Generated scenarios are lists of global events. The minimization algorithm (see chapters 3 and 4) is applied on such sequences which means that:

- it can produce sequences like $(\langle a, b, c\rangle, a)$ where two global events having common elementary events appear (here the vector $\langle a, b, c\rangle$ and a alone).
- if $(\langle a, b, c\rangle)$ and (a) are two generated sequences (i.e that produces the diagnosis objective), the former is not removed by the minimization algorithm that consider both events as unrelated.

Chapter 3

Residual Language Decision Diagrams

In this chapter we present a data structure defined in [13] similar to Zero-suppressed BDDs (ZBDD) introduced by S. Minato[12]. While ZBDDs permit to store finite sets of subsets, RLDDs encode finite sets of words. This data structure allows classical operations over sets like union or intersection but also a minimization operation that computes minimal sequences (for the sub-word order). Recently this data structure has been studied in the context of data-mining under the name of *SeqBDD*[7, 11].

3.1 Residual languages and RLDDs

In the sequel we consider a finite *alphabet* Σ. A *word* w is a sequence $a_1 a_2 \dots a_n$ of letters a_i belonging to Σ. The number of letters in a word is its *length*; we denote by ϵ the word of length 0. The set of words over Σ built with n letters is denoted Σ^n. The set of all words (ϵ included) over Σ is denoted $\Sigma^\star$. A *language* over Σ is any subset of $\Sigma^\star$.

The concatenation of two words w_1, w_2 taken in $\Sigma^\star$ is denoted $w_1.w_2$; ϵ is the identity element for concatenation. If $w \in \Sigma^\star$ is a word and $L \subseteq \Sigma^\star$ a language then $w.L$ is the language $\{w.x \mid x \in L\}$ i.e., the set of words formed by w followed by any word belonging to L.

For any language $L \subseteq \Sigma^\star$ and any letter $a \in \Sigma$, the *residual language of L by a* is the language denoted by $a^{-1}L$ and defined by:

$$a^{-1}L = \{u \in \Sigma^\star \mid a.u \in L\}$$

The reader should notice that $a^{-1}L$ is not necessarily a subset of L and, as shown on following example, $a.(a^{-1}L)$ is not always L but the set of residual languages of L is a partition of L: $L = \cup_{a\in\Sigma} a.(a^{-1}L)$.

Example 2 *Let* $\Sigma = \{a, b, c, d\}$ *and* $L = \{abc, aabc, bac, ca\}$. *Then:* $a^{-1}L = \{bc, abc\}$, $b^{-1}L = \{ac\}$, $c^{-1}L = \{a\}$, $d^{-1}L = \emptyset$.

In the sequel we will use the following notations:

- $L_a = a.(a^{-1}L)$ the subset of L whose words start with letter a.
- $L_{\overline{a}} = L \setminus L_a$ the words of L that do not start with a.

L_a and $L_{\overline{a}}$ is a partition of L i.e $L_a \cap L_{\overline{a}} = \emptyset$ and $L_a \cup L_{\overline{a}} = L$. This decomposition of L is used to defined a data structure to store any finite language L in a compact way. The previous partitioning plays the same role as the well-known *Shannon decomposition theorem* used to build BDDs[2].

We have called this data structure *RLDD* for *Residual Language Decision Diagram*. As other decision diagrams, a RLDD is a directed acyclic graph (DAG).

Definition 2 (RLDD) *A* RLDD *is a DAG with three kind of nodes:*

- *either one of the two leaves* $\mathbf{0}$ *or* $\mathbf{1}$
- *or an intermediate node denoted* $N = \langle a, N_1, N_2 \rangle$ *where* $a \in \Sigma$ *is the label of the node and,* N_1 *and* N_2 *are children nodes of* N.

Definition 3 (Height) *The* height $H(N)$ *of a RLDD* N *is defined inductively by:*

- $H(\mathbf{0}) = H(\mathbf{1}) = 0$
- $H(\langle a, N_1, N_2 \rangle) = 1 + max(H(N_1), H(N_2))$

Definition 4 (Semantics) *The semantics of a RLDD* N *is a language denoted* $[\![N]\!] \subseteq \Sigma^\star$ *defined recursively on the structure of RLDDs:*

- $[\![\mathbf{0}]\!] = \emptyset$
- $[\![\mathbf{1}]\!] = \{\epsilon\}$
- $[\![\langle a, N_1, N_2 \rangle]\!] = a.[\![N_1]\!] \cup [\![N_2]\!]$

A RLDD encoding the language L given in example 2 is depicted on figure 3.1. When represented graphically, the edge between an intermediate node and its left child is decorated with a black bullet $\bullet$ (to recall concatenation operation).

Depending on letters used to decompose successive residuals the RLDD encoding a language L is different. For instance the figure 3.2 depicts a RLDD encoding the same language than the one on figure 3.1. While on the first figure (3.1) letters were always selected in the order b, a and c, on the second one, letters were chosen in the order a, b and c.

To ensure canonicity of the representation and to improve efficiency of algorithms we have to enforce the choice of the letter used to decompose a language. In the sequel we assume that Σ is an alphabet completely ordered by

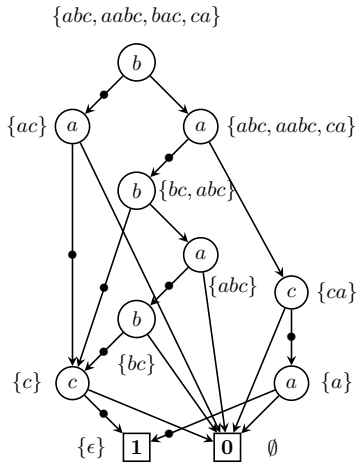


Figure 3.1: A RLDD encoding $L = \{abc, aabc, bac, ca\}$.

some relation $<$. The letter $\rho(L)$ used to decompose a language L is defined according to this order as the smallest letter that gives a non-empty residual language:

Definition 5 (Decomposition letter) *For any non-empty language L that is not the singleton $\{\epsilon\}$, we define:*

$$\rho(L) = min_{<}(\{a \in \Sigma \mid a^{-1}L \neq \emptyset\})$$

For any finite language $L \subseteq \Sigma^\star$, the canonical RLDD representing L, $\texttt{RLDD}(L)$ is defined by:

- $\texttt{RLDD}(L) = \mathbf{0}$ if L is empty;
- $\texttt{RLDD}(L) = \mathbf{1}$ if L is $\{\epsilon\}$;
- $\texttt{RLDD}(L) = \langle a, \texttt{RLDD}(a^{-1}L), \texttt{RLDD}(L_{\overline{a}})\rangle$ where $a = \rho(L)$.

Example 3 *We come back on the language given in example 2. Now we consider that Σ is ordered as follows: $a < b < c$. The canonical RLDD encoding L is given on the figure 3.2. This RLDD has been built as follows:*

- $\lambda_0 = \texttt{RLDD}(\{abc, aabc, bac, ca\}) = \langle a, \texttt{RLDD}(\{bc, abc\}), \texttt{RLDD}(\{bac, ca\})\rangle = \langle a, \lambda_1, \lambda_2\rangle$
- $\lambda_1 = \texttt{RLDD}(\{bc, abc\}) = \langle a, \texttt{RLDD}(\{bc\}), \texttt{RLDD}(\{bc\})\rangle = \langle a, \lambda_3, \lambda_3\rangle$
- $\lambda_2 = \texttt{RLDD}(\{bac, ca\}) = \langle b, \texttt{RLDD}(\{ac\}), \texttt{RLDD}(\{ca\})\rangle = \langle b, \lambda_4, \lambda_5\rangle$
- $\lambda_3 = \texttt{RLDD}(\{bc\}) = \langle b, \texttt{RLDD}(\{c\}), \emptyset\rangle = \langle b, \lambda_6, \mathbf{0}\rangle$
- $\lambda_4 = \texttt{RLDD}(\{ac\}) = \langle a, \texttt{RLDD}(\{c\}), \emptyset\rangle = \langle a, \lambda_6, \mathbf{0}\rangle$

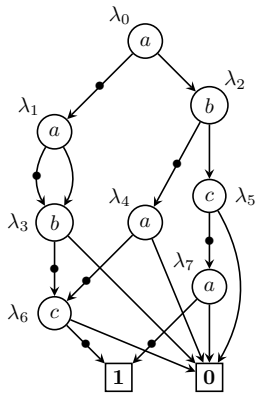


Figure 3.2: RLDD encoding the same language that the RLDD on figure 3.1. The order $a < b < c$ is used.

- $\lambda_5 = \texttt{RLDD}(\{ca\}) = \langle c, \texttt{RLDD}(\{c\}), \emptyset\rangle = \langle c, \lambda_7, \mathbf{0}\rangle$
- $\lambda_6 = \texttt{RLDD}(\{c\}) = \langle c, \texttt{RLDD}(\{\epsilon\}), \emptyset\rangle = \langle c, \mathbf{1}, \mathbf{0}\rangle$
- $\lambda_7 = \texttt{RLDD}(\{a\}) = \langle a, \texttt{RLDD}(\{\epsilon\}), \emptyset\rangle = \langle a, \mathbf{1}, \mathbf{0}\rangle$

3.2 Basic operations on RLDDs

In this section we present three basic operations that are required to compute minimal words. These operations, *build*, *union* and *intersection*, are defined recursively on the structure of their operands. In the sequel, if $N = \langle a, N_1, N_2\rangle$ is a RLDD we denote:

- $\lambda(N)$ its label a;
- $\alpha(N)$ its left child N_1;
- and $\beta(N)$ its right child N_2.

The `build` operator is used to ensure that left child is not empty.

Build: $\texttt{build} : \Sigma \times RLDD \times RLDD \longrightarrow RLDD$

- $\texttt{build}(a, \mathbf{0}, Y) = Y$ for any RLDD Y;
- $\texttt{build}(a, X, Y) = \langle a, X, Y\rangle$ if $X \neq \mathbf{0}$

Union $\texttt{union} : RLDD \times RLDD \longrightarrow RLDD$

- $\texttt{union}(X, X) = X$ for any RLDD X
- $\texttt{union}(\mathbf{0}, X) = \texttt{union}(X, \mathbf{0}) = X$ for any RLDD X
- $\texttt{union}(\mathbf{1}, N) = \texttt{union}(N, \mathbf{1}) = \langle \lambda(N), \alpha(N), \texttt{union}(\mathbf{1}, \beta(N)) \rangle$
- $\texttt{union}(N_1, N_2) = \langle a, N, N' \rangle$ where:
  - $a = min(\lambda(N_1), \lambda(N_2))$
  - if $\lambda(N_1) = \lambda(N_2)$ then $N = \texttt{union}(\alpha(N_1), \alpha(N_2))$ and $N' = \texttt{union}(\beta(N_1), \beta(N_2))$
  - if $\lambda(N_1) < \lambda(N_2)$ then $N = \alpha(N_1)$ and $N' = \texttt{union}(\beta(N_1), N_2)$
  - if $\lambda(N_1) > \lambda(N_2)$ then $N = \alpha(N_2)$ and $N' = \texttt{union}(N_1, \beta(N_2))$

Intersection $\texttt{inter} : RLDD \times RLDD \longrightarrow RLDD$

- $\texttt{inter}(X, X) = X$ for any RLDD X
- $\texttt{inter}(\mathbf{0}, X) = \texttt{inter}(X, \mathbf{0}) = \mathbf{0}$ for any RLDD X
- $\texttt{inter}(\mathbf{1}, N) = \texttt{inter}(N, \mathbf{1}) = \texttt{inter}(\mathbf{1}, \beta(N))$
- $\texttt{inter}(N_1, N_2) = N$ where:
  - if $\lambda(N_1) = \lambda(N_2)$ then $N = \texttt{build}(\lambda(N_1), \texttt{inter}(\alpha(N_1), \alpha(N_2)), \texttt{inter}(\beta(N_1), \beta(N_2)))$
  - if $\lambda(N_1) < \lambda(N_2)$ then $N = \texttt{inter}(\beta(N_1), N_2)$
  - if $\lambda(N_1) > \lambda(N_2)$ then $N = \texttt{inter}(N_1, \beta(N_2))$

Theorem 1 *For any letter $a \in \Sigma$ and any RLDDs X and Y,*

- $[\![\texttt{build}(a, X, Y)]\!] = a.[\![X]\!] \cup [\![Y]\!]$
- $[\![\texttt{union}(X, Y)]\!] = [\![X]\!] \cup [\![Y]\!]$
- $[\![\texttt{inter}(X, Y)]\!] = [\![X]\!] \cap [\![Y]\!]$

Proof: By induction on $H(X) + H(Y)$. $\Diamond$

3.3 Minimal words

Given a set of words seen as critical scenarios, several criteria can be imagined to determine which words are the most representatives. In the sequel we will denote by $u \sqsubseteq v$ the fact that u is more important than v.

The simplest criterion is certainly the length of words but it is too strong. Of course it selects shortest scenarios so the most important ones, but it completely ignores qualitative informations of sequences. For instance, if a and

bc are two failure sequences that involve separate parts of the system, bc is rejected.

A second criterion could be the prefix order. We consider that a sequence $u \sqsubseteq v$ if u starts (or is a prefix of) v. This criterion mixes length and qualitative informations. Actually if u is a prefix of v, u is more probable than v and taking countermeasures to prevent u should resolve issues for v. However, the criterion is too weak because it keeps useless sequences. For instance, abc and bc can not be compared with the prefix order; so both are kept while, clearly, abc should be forgotten. In this case, a comparison of suffixes should be preferred.

The factors can also be used. $u \sqsubseteq v$ if $v = x.u.y$ where x and y are arbitrary and possibly empty words; u is said to be a *factor* of v. This criterion gathers advantages of length, prefixes, suffixes and handles what happens in the "middle" of sequences. But factors ignore the case where letters of u are dispatched into v i.e. for instance abc and ac.

Finally a fourth criterion will be used and, intuitively it is very similar to the one used for cuts (i.e. inclusion). For now on, we consider that $u \sqsubseteq v$ if u is a sub-word of v. More formally, we define this order as follows.

Definition 6 (Subwords) *Let $u = u_1 \dots u_n$ and $v = v_1 \dots v_m$ be two words of length n and m respectively. u is a* sub-word *of v if there exists a mapping $\sigma : [1, n] \to [1, m]$ such that:*

1. *σ is strictly increasing i.e. for any $1 \leq i < j \leq n$, $\sigma(i) < \sigma(j)$;*
2. *$\forall i \in [1, n]$, $u_i = v_{\sigma(i)}$*

The above definition simply describes the fact that if u is a subword of v ($u \sqsubseteq v$) then letters of u can be dispatched into v with respect of their order in u.

3.3.1 A decomposition theorem for sub-words

The computation of minimal words of L is based on a simple idea:

1. First, L is split into two disjoint subsets, say X and Y;
2. Then, we remove from X words that have a sub-word in Y and conversely, we remove from Y words that have a sub-word in X.
3. Finally, we join together obtained sets to get minimal words $min_{\sqsubseteq}(L)$.

For the first step, a partition of language L is obtained easily using residual languages. If $a = \rho(L)$ then we choose to split L as L_a and $L_{\overline{a}}$. The second step requires the introduction of a new operation over languages; we denote this operation $\div$ (as in [16]). This operation removes from a language X all words that are not minimal with respect to words of a second language Y. More formally, $\div$ is defined by:

$$\forall X, Y \subseteq \Sigma^\star, X \div Y = \{x \in X \mid \forall y \in Y, y \not\sqsubseteq x\}$$

This operation $\div$ has the following properties for all subsets X, Y and Z of $\Sigma^\star$ and letters $a, b \in \Sigma$:

1. $X \div \emptyset = X$
2. $X \div \{\epsilon\} = \emptyset$
3. $(X \cup Y) \div Z = X \div Z \cup Y \div Z$
4. $X \div (Y \cup Z) = X \div Y \cap X \div Z$
5. $a.X \div a.Y = a.(X \div Y)$
6. $a \neq b \Rightarrow a.X \div b.Y = a.(X \div b.Y)$
7. $X \div Y = [(X_a \div Y_a) \cap (X_a \div Y_{\overline{a}})] \cup (X_{\overline{a}} \div Y)$

Lemma 1 $\forall L \subseteq \Sigma^\star \backslash \{\epsilon\}, \forall a \in \Sigma, \forall w \in \Sigma^\star, a.w \in min(L) \iff w \in min(a^{-1}L) \div L_{\overline{a}}$

Proof: Assume that $a.w \in min(L)$. We have $w \in a^{-1}L$. There is no $v \in a^{-1}L$ such that $v \sqsubset w$ because we would have $a.v \sqsubset a.w$ which refutes $a.w \in min(L)$; thus, $w \in min(a^{-1}L)$. If $v \in L_{\overline{a}}$ is such that $v \sqsubset w$ yields the same contradiction because, in this case, $v \sqsubset a.w$.

For the converse, let $w \in min(a^{-1}L) \div L_{\overline{a}}$ and $v \in L$ such that $v \sqsubseteq a.w$. If $v = a.x$ with $x \in \Sigma^\star$, we have $x \sqsubset w$ which contradicts $w \in min(a^{-1}L)$. If $v \in L_{\overline{a}}$ i.e $v = b.x$ with $x \in \Sigma^\star$, we have $b.x \sqsubset w$ but it is impossible because w would have been removed from $min(a^{-1}L)$ by $\div$. $\Diamond$

Lemma 2 $\forall L \subseteq \Sigma^\star \setminus \{\epsilon\}, \forall a, b \in \Sigma, \forall w \in \Sigma^\star, a \neq b \Rightarrow (b.w \in min(L) \iff b.w \in min(L_{\overline{a}}) \div L_a)$

Proof: Let $b.w \in min(L)$; clearly $b.w \in min(L_{\overline{a}})$. The hypothesis imposes that for any $a.x \in L_a$, $a.x \not\sqsubset b.w$; so $b.w \in min(L_{\overline{a}}) \div L_a$. For the converse, assume $b.w \in min(L_{\overline{a}}) \div L_a$ and $x \in L$ such that $x \sqsubset b.w$. x can not be in L_a because $b.w$ would have been removed from $min(L_{\overline{a}})$ by $\div$. x can neither be in $L_{\overline{a}}$ because $b.w \in min(L_{\overline{a}})$. We can conclude that for $x \in L$, $x \not\sqsubset b.w$ and, thus, $b.w \in min(L)$. $\Diamond$

Corollary 1 (Decomposition of sets of minimal words)

$$\forall L \subseteq \Sigma^\star, \forall a \in \Sigma, min(L) = a.(min(a^{-1}L) \div L_{\overline{a}}) \cup (min(L_{\overline{a}}) \div L_a)$$

The previous theorem shows that computation of minimal words can follows the structure of languages given by their residual languages. This result permits us to easily design an algorithm computing $min_{\sqsubseteq}$ based on the RLDD data structure.

3.3.2 Application to RLDDs

In order to compute minimal sequences we have to define $\div$ and min for RLDDs. As others, these operations, called respectively `div` and `minimize`, are defined according to the RLDD structure:

Extraction $\texttt{div} : RLDD \times RLDD \longrightarrow RLDD$

- $\texttt{div}(\mathbf{0}, X) = \mathbf{0}$ for any RLDD X
- $\texttt{div}(X, \mathbf{0}) = X$ for any RLDD X
- $\texttt{div}(X, \mathbf{1}) = \mathbf{0}$ for any RLDD X
- $\texttt{div}(\mathbf{1}, N) = \texttt{div}(\mathbf{1}, \beta(N))$
- $\texttt{div}(N, N') = \texttt{build}(\lambda(N), N_1, N_2)$ where:
  - if $\lambda(N) = \lambda(N')$, $N_1 = \texttt{inter}(\texttt{div}(\alpha(N), \alpha(N')), \texttt{div}(\alpha(N), \beta(N')))$,
  - if $\lambda(N) \neq \lambda(N')$, $N_1 = \texttt{div}(\alpha(N), N')$
  - $N_2 = \texttt{div}(\beta(N), N')$

Theorem 2 *For any RLDDs N, $[\![\texttt{div}(N, N')]\!] = [\![N]\!] \div [\![N']\!]$*

Proof: We show the result by induction on $n = H(N) + H(N')$. We consider only the last case. We assume the property satisfied for any $k < n$. Let N and N', two RLDDs such that $H(N) + H(N') = n$. Let $a = \lambda(N)$ and $a' = \lambda(N')$. We consider the two cases;

$\mathbf{a = a'}$: By construction we have:

$$[\![\texttt{div}(N, N')]\!] = a.([\![\texttt{div}(\alpha(N), \alpha(N'))]\!] \cap [\![\texttt{div}(\alpha(N), \beta(N'))]\!]) \cup [\![\texttt{div}(\beta(N), N')]\!]$$

Then, induction hypothesis gives us:

$$\begin{aligned}[\![\texttt{div}(N, N')]\!] &= a.([\![\alpha(N)]\!] \div [\![\alpha(N')]\!] \cap [\![\alpha(N)]\!] \div [\![\beta(N')]\!]) \cup ([\![\beta(N)]\!] \div [\![N']\!]) \\ &= a.([\![\alpha(N)]\!] \div [\![\alpha(N')]\!]) \cap a.([\![\alpha(N)]\!] \div [\![\beta(N')]\!]) \cup ([\![\beta(N)]\!] \div [\![N']\!])\end{aligned}$$

Using properties 5 and 6 of $\div$ we get:

$$\begin{aligned}[\![\texttt{div}(N, N')]\!] &= (a.[\![\alpha(N)]\!] \div a.[\![\alpha(N')]\!]) \cap (a.[\![\alpha(N)]\!] \div [\![\beta(N')]\!]) \cup ([\![\beta(N)]\!] \div [\![N']\!]) \\ &= (a.[\![\alpha(N)]\!] \div (a.[\![\alpha(N')]\!] \cup [\![\beta(N')]\!])) \cup ([\![\beta(N)]\!] \div [\![N']\!]) \\ &= (a.[\![\alpha(N)]\!] \div [\![N']\!]) \cup ([\![\beta(N)]\!] \div [\![N']\!]) \\ &= (a.[\![\alpha(N)]\!] \cup [\![\beta(N)]\!]) \div [\![N']\!] \\ &= [\![N]\!] \div [\![N']\!]\end{aligned}$$

$\mathbf{a \neq a'}$: We use the same method but this time we have:

$$[\![\texttt{div}(N, N')]\!] = a.([\![\texttt{div}(\alpha(N), N')]\!]) \cup [\![\texttt{div}(\beta(N), N')]\!]$$

Then, induction hypothesis gives us:

$$\begin{aligned}[\![\texttt{div}(N, N')]\!] &= a.([\![\alpha(N)]\!] \div [\![N']\!]) \cup ([\![\beta(N)]\!] \div [\![N']\!]) \\ &= (a.[\![\alpha(N)]\!] \div [\![N']\!]) \cup ([\![\beta(N)]\!] \div [\![N']\!]) \\ &= (a.[\![\alpha(N)]\!] \cup [\![\beta(N)]\!]) \div [\![N']\!]) \\ &= [\![N]\!] \div [\![N']\!]\end{aligned}$$

$\Diamond$

Minimization $\texttt{minimize} : RLDD \longrightarrow RLDD$

- $\texttt{minimize}(\mathbf{0}) = \mathbf{0}$
- $\texttt{minimize}(\mathbf{1}) = \mathbf{1}$
- $\texttt{minimize}(N) = \texttt{build}(\lambda(N), N_1, N_2)$ where:
  - $N_1 = \texttt{div}(\texttt{minimize}(\alpha(N)), \beta(N))$
  - $N_2 = \texttt{div}(\texttt{minimize}(\beta(N)), \texttt{build}(\lambda(N), \alpha(N), \mathbf{0}))$

Theorem 3 *For any RLDD N, $[\![\texttt{minimize}(N)]\!] = min([\![N]\!])$*

Proof: Straightforwards by induction on the structure of RLDDs. $\Diamond$

3.4 Complexity

In above section, algorithms are specified using recursive equations based on the structure of RLDDs. Applying these algorithms as-is could yield execution times exponential into the height of diagrams. This exponential cost comes from multiple recursive calls that visit several times a same path. The well-known technique of computation caches (e.g. [3]) must be used to reduced the number of redundant visits of paths.

Unfortunately `div` operation can yield diagrams with an exponential number of nodes while its arguments have a polynomial size. This cost can not be absorbed by cache techniques. For instance, if Σ is an alphabet with n letters, then

$$U_n = \{u \in \Sigma^n \mid \forall a \in \Sigma, |u|_a = 1\}$$

where $|u|_a$ is the number of letters a in u, is a language that requires $n2^{n-1} + 2$ nodes to be represented by a RLDD. U_n is the set of words composed of n distinct letters. U_n can be obtained by applying $\div$ to two languages that are encoded with a polynomial number of nodes:

1. Σ^n the set of words of length n that requires $n^2 + 2$ nodes;
2. $Sqr(\Sigma) = \{a.a \mid a \in \Sigma\}$ the set of words of length 2 containing twice the same letter. This language requires $2n + 2$ nodes.

We have $U_n = \Sigma^n \div Sqr(\Sigma)$ because $Sqr(\Sigma)$ removes from Σ^n words with multiple occurrences of same letters. Figures 3.3, 3.4 and 3.5 depict respectively Σ^n, $Sqr(\Sigma)$ and U_n for an alphabet Σ containing 3 letters.

The size of U_n in terms of RLDD nodes, $n2^{n-1} + 2$, has been obtained experimentally but it can be proved that the same phenomenon also holds for minimal automata that recognize Σ^n, $Sqr(\Sigma)$ and U_n which have, respectively, $n + 1$, $n + 2$ and 2^n states[1].

In practice, in the context of failure scenarios computation, such diagrams should not be problematic because languages (i.e. sets of scenarios) would contain few (around one hundred words) and short words (less than ten events).

[1] plus one if we count sink state.

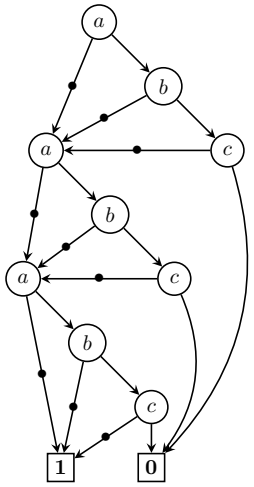


Figure 3.3: A RLDD encoding $\{a, b, c\}^3$

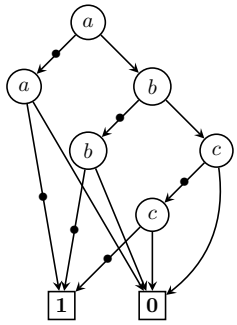


Figure 3.4: A RLDD for $Sqr(\{a, b, c\})$.

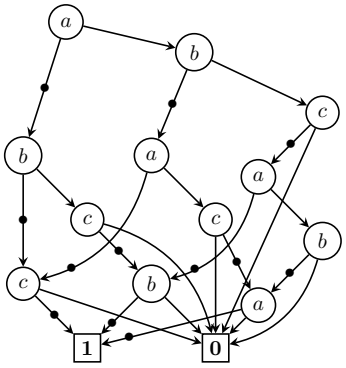


Figure 3.5: A RLDD encoding $U_3 = \{a, b, c\}^3 \div Sqr(\{a, b, c\})$

Chapter 4

Minimal cuts and minimal sequences

This chapter presents algorithms used to minimize sets of sequences and sets of cuts computed using algorithms presented in chapter 2.

Both algorithms (for sequences and cuts) take as input a Decision Diagram[5] representing all computed scenarios that yield an unexpected configuration. Minimization algorithms are based on RLDDs (see chapter 3). However, since the encoding of scenarios by means of DDs is not the same in both cases, we have to write two specific translation algorithms.

4.1 A brief introduction to DDs

Decision Diagrams (DD) used by ARC[9] model-checker, are stemmed from the TOUPIE tool[5]. DDs are diagrams built over non-Boolean variables. Domains of variables remain finite but they can contain an arbitrary number of elements. The principle is the same than for BDDs but, instead of taking the decision according to two values we use N values (if N is the cardinality of the domain of the variable).

DDs are used in ARC to represent sets of configurations, relation transitions or more generally relations. Figure 4.1 gives an example of a DD that represents reachable configuration of an AltaRica node.

In the context of scenarios generation, variables are added to AltaRica nodes to record occurrences of events. After the computation of reachable configurations, the DD is projected on these additional variables (i.e. all other variables are removed) to keep only data related to expected scenarios. Depending on the nature of computed objects, sets or words, resulting DDs have different semantics and different structures. Figure 4.2 depicts DDs that encode the same scenarios abc, ac and b but in a one case they are viewed as cuts and in the other case, as sequences.

```
node A
  flow
    i, o : [0, 2];
  state
    s : { ok, nok };
  event
    fail;
  trans
    s = ok |- fail -> s := nok;
  assert
    o = (if s = ok then i else 0);
edon
```

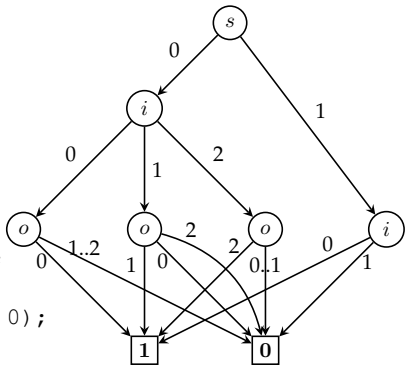


Figure 4.1: An AltaRica node and the DD encoding its reachable configurations. Symbolic values `ok` and `nok` are encoded by integer values 0 and 1.

4.2 Sequence and minimal sequences

As mentioned in introduction, minimization algorithm is based on RLDDs. We could design a direct computation of minimal sequences from the DD that encodes sequences but we have preferred to generate an intermediate structure that permits to easily manipulate and display the whole set of sequences.

The main issue is thus to translate the DD that encodes computed sequences into a RLDD and then to apply the minimization operator defined over RLDDs (see 3.3). This translation step is straightforward if p_is variables that memorize i^{th} events are ordered according to i in the DD. If this is not the case a simple relabelling of variables is realized. Under the hypothesis of a good ordering of variables, algorithm 2 (page 37) translates a DD into a RLDD. In the pseudo-code of algorithm 2 we have use suffixes DD and RL to distinguish trivial nodes (**0** and **1**) of DDs and RLDDs.

Due to the encoding of sequences by p_i variables, the DD can be viewed as a classical automaton recognizing words where $\mathbf{1}_{DD}$ is the accepting state, $\mathbf{0}_{DD}$ is a sink state and 0-labelled edges are interpreted as ϵ transitions. The algorithm recursively visits paths to the accepting state (i.e the leaf node $\mathbf{1}_{DD}$) using a depth-first search traversal of the input DD.

As usual a cache C is used (lines 7 and 17) to prevent useless revisit of same paths of the DD; if the cache does not erase its entries the number of recursive calls is linear in the number of nodes of the DD.

Figure 4.3 shows the application of algorithm 2 on the DD depicted on the right side of figure 4.2. Note that, on this example the cache is useless; except leaves, all DD nodes have only one input edge.

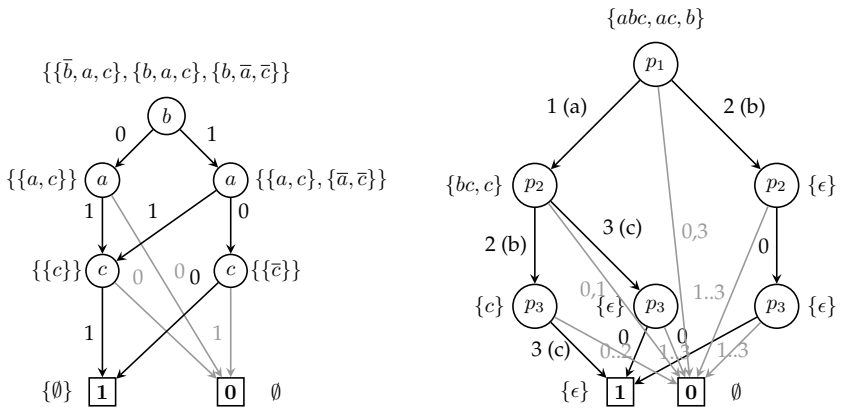


Figure 4.2: Different representations of scenarios abc, ac and b by means of DDs. On the left hand side, scenarios are considered as *cuts* (note that in this case, $\overline{b}$ and $\overline{a}\overline{c}$ are implicit in ac and b). Each variable of the DD corresponds to an event and are ordered as $b < a < c$. Edges to leaf node **0** are drawn in gray. On the right hand side, scenarios are viewed as *sequences* where $a = 1$, $b = 2$ and $c = 3$ (recall observer described in chapter 2). Length of sequences is bound by $k = 3$. Note that for sequences with less than k events, scenarios are padded with 0s.

4.3 Minimal cuts

The algorithm is inspired by the one proposed by A. Rauzy in [16]. The original algorithm (section C of [16]) computes for a Boolean formula F a ZBDD[12] that encodes minimal cuts of F according to a set L of *significant* literals. Literals can be positive and/or negative occurrences of elementary events. That means that non-failures could be considered as relevant informations.

In the context of this study we have chosen to ignore these negative informations and to handle only positive events (i.e actual failures). This hypothesis corresponds to case 2 of Rauzy's algorithm where, for all failure event e, $e \in L$ and $\overline{e} \notin L$. Algorithm 3 is an implementation of Rauzy's algorithm (case 2) using RLDDs instead of ZBDDs. Remember that DDs that encode cuts have at most two children nodes because the observer uses only Boolean variables. The function `variable`(N) returns the index of the variable labelling the node. `cofactor`(N, i) returns the i^{th} child of N.

Figure 4.4 depicts application of algorithm 3 on the DD depicted on the left side of figure 4.2.

The main difference between Rauzy's algorithm and algorithm 3 is the use of `rldd-div` instead of the similar operation $\div$ defined in [8] for ZBDDs. This replacement can be easily justified by the fact that due to ordering of variables, cuts are implicitly treated as words. More precisely, if for any cut σ, we denote by $w(\sigma)$ the word obtained by concatenating elements of σ according to the

Algorithm 2 `dd-to-rldd` (DD N, cache C)

```
1: RLDD R /* the result */
2: if N = 0_DD then
3:    R = 0_RL
4: else if N = 1_DD then
5:    R = 1_RL
6: else
7:    R = cache-find-operation(C, dd-to-rldd, N)
8:    if R = NULL then
9:       R = 0_RL
10:      for all outgoing edge N --a--> N' do
11:         RLDD son = dd-to-rldd(N', C)
            /* a = 0 means that current p_i was not assigned i.e the
            edge is treated as an ε.  */
12:         if a ≠ 0 then
13:            son = rldd-build(a, son, 0_RL)
14:         end if
15:         R = rldd-union(R, son)
16:      end for
17:      cache-memorize-operation(C, dd-to-rldd, N, R)
18:   end if
19: end if
20: return R
```

order used to build DDs, then for any cuts σ_1 and σ_2, $\sigma_1 \subseteq \sigma_2$ if and only if $w(\sigma_1)$ is a sub-word of $w(\sigma_2)$.

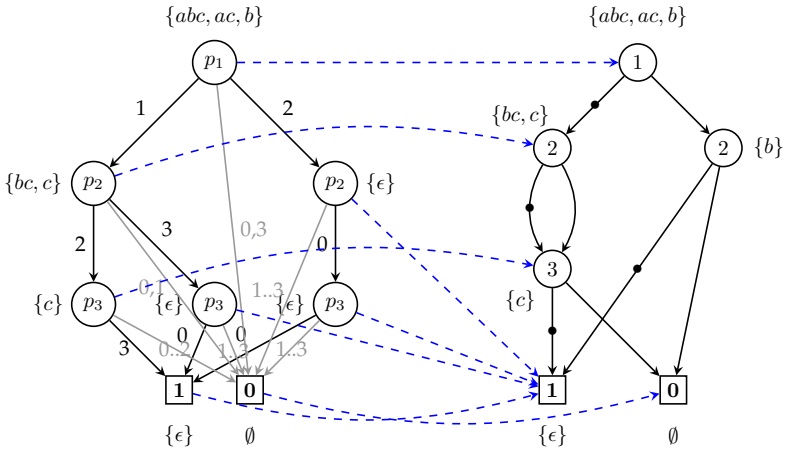


Figure 4.3: Application of the `dd-to-rldd` algorithm onto the DD encoding sequences abc, ac and b depicted on figure 4.2. Dashed arrows show the mapping realized by the algorithm between DD (on the left) and RLDD nodes (on the right). Remember that letters a, b and c are encoded respectively by integers 1, 2 and 3. On DDs, letters label edges while on RLDDs, letters label nodes.

Algorithm 3 `dd-to-min-rldd` (DD N, cache C)

```
1: RLDD R /* the result */
2: if N = 0_DD then
3:    R = 0_RL
4: else if N = 1_DD then
5:    R = 1_RL
6: else
7:    R = cache-find-operation(C, dd-to-min-rldd, N)
8:    if R = NULL then
9:       int v = variable(N)
10:      DD N_0 = cofactor(N, 0)
11:      DD N_1 = cofactor(N, 1)
12:      RLDD son0 = dd-to-min-rldd(N_0, C)
13:      RLDD son1 = dd-to-min-rldd(N_1, C)
14:      RLDD tmp = rldd-div(son1, son0)
15:      R = rldd-build(v, tmp, son0)
16:      cache-memorize-operation(C, dd-to-min-rldd, N, R)
17:   end if
18: end if
19: return R
```

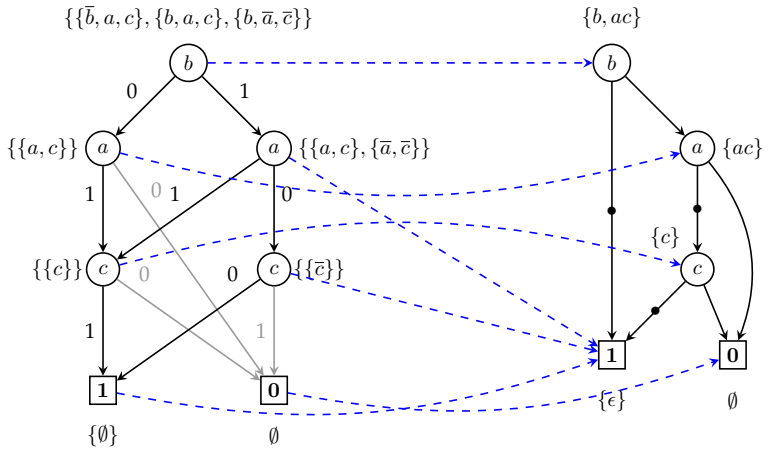


Figure 4.4: Application of the `dd-to-min-rldd` algorithm onto the DD encoding cuts abc, ac and b depicted on figure 4.2. Dashed arrows show the mapping realized by the algorithm between DD (on the left) and RLDD nodes (on the right). Variables are ordered as follows: $b < a < c$.

Chapter 5

Reduction of models

5.1 Principles of the reduction

The reduction algorithm is applied to the *constraint automaton*[14] that represents the model. This automaton, actually an AltaRica node, is obtained using a rewriting process that removes all hierarchical levels of the model and compiles this latter into an equivalent n ode. This process has been called the *flat semantics* of the model[13].

Given a constraint automaton $\mathcal{A}$ that represents the system we want to evaluate a Boolean formula ϕ built over variables of $\mathcal{A}$. ϕ will be called *target formula*. Our goal is to reduce $\mathcal{A}$ to its only parts (i.e variables, assertions and transitions) that are mandatory to evaluate ϕ all along runs (or behaviours) of the automaton.

More precisely the restricted automaton, denoted $\mathcal{A}^\phi$, should satisfy following properties:

1. For any configuration σ of $\mathcal{A}$ one can associate its projection σ' in $\mathcal{A}^\phi$ and σ satisfies ϕ iff σ' satisfies ϕ.

2. For all executable sequences of macro-transitions of $\mathcal{A}$, its restriction to macro-transitions belonging to $\mathcal{A}^\phi$ is actually an executable sequence in $\mathcal{A}^\phi$.

3. Any executable sequence of macro-transitions of $\mathcal{A}^\phi$ is an executable sequence of $\mathcal{A}$.

The main idea that underlies the algorithm is to compute items that may influence the truth value of ϕ. Starting from variables appearing in ϕ, one computes elements of $\mathcal{A}$ related to these variables: flow variables are influenced by assertions, assertions by flow and state variables, state variables by transitions and so on. These relations between components of $\mathcal{A}$ permit to restrict $\mathcal{A}$ to a subset of its variables; other parts not related to these variables are simply ignored.

5.2 Constraint Automata

If V is a set of variables, $T(V)$ and $BF(V)$ denote, respectively, sets of terms and sets of Boolean formulae built over a set of variables V. For the sake of simplicity we shall assume that variables take their values into a unique domain $\mathcal{D}$. If e is a term or a Boolean formula, we denote by $Var(e)$ the set of variables occurring in e. If $f : A \rightarrow B$ is a function from A into B, we denote by $Dom(f) \subseteq A$ its domain. $F(A, B)$ will note the set of functions from A into B.

A *valuation* of variables of V is a mapping belonging to $F(V, \mathcal{D})$. If $V' \subseteq V$ and $\sigma \in F(V, \mathcal{D})$ then we denote by $\sigma[V'] \in F(V', \mathcal{D})$ the restriction of the valuation σ to variables belonging to V'. If $F \in BF(V)$, we denote by $[\![F]\!]$ the set of valuations that satisfy F. If $t \in T(V)$, $[\![t]\!]$ is the function from $F(V, \mathcal{D})$ into $\mathcal{D}$ that evaluates a term w.r.t to a given valuation of variables.

Definition 7 (Constraint Automaton) *A constraint automaton $\mathcal{A} = \langle V_S, V_F, E, T, A, I\rangle$ is a tuple where:*

- *V_S and V_F are disjoint sets of variables. Elements of V_S (resp. V_F) are called* state *(resp.* flow*) variables.*
- *E is a set of events containing a distinguished event ϵ.*
- *$T \subseteq BF(V_S \cup V_F) \times E \times F(V_S, T(V_S \cup V_F))$ is the set of* macro-transitions*. If $\langle g, e, \alpha\rangle$ belongs to T, g is called the* guard*, e the event and α the assignment of the transition. α is not necessarily total. We assume that T contains the "don't change state" macro-transition $\langle \mathtt{true}, \epsilon, \emptyset\rangle$.*
- *$A \subseteq BF(V_S \cup V_F)$ contains* assertions *of the model, i.e., invariants that must be satisfied by valuations of variables.*
- *$I \subseteq BF(V_S)$ contains initial constraints.*

A *configuration* of $\mathcal{A}$ is a valuation of its variables satisfying A (in the sequel we do not distinguish A or I and the conjunction of their elements).

If σ_1 and σ_2 are two configurations of $\mathcal{A}$ and if $t = \langle g, e, \alpha\rangle \in T$, then we denote by $\sigma_1 \xrightarrow{t}_{\mathcal{A}} \sigma_2$ the fact that t is *enabled* in configuration σ_1 and change the configuration of $\mathcal{A}$ into σ_2; or more formally we have:

- $\sigma_1 \in [\![A]\!]$, $\sigma_2 \in [\![A]\!]$
- $\sigma_1 \in [\![g]\!]$
- $\forall v \in V_S \cap Dom(\alpha)$, $\sigma_2(v) = [\![\alpha(v)]\!](\sigma_1)$
- $\forall v \in V_S \setminus Dom(\alpha)$, $\sigma_2(v) = \sigma_1(v)$

Words belonging to $T^\star$ shall denote sequences of macro-transitions. If $w = t_1 \dots t_n \in T^\star$ then we will write $\sigma_0 \xrightarrow{w}_{\mathcal{A}} \sigma_n$ if there exist configurations $\sigma_1, \dots,$

σ_{n-1} such that $\sigma_0 \xrightarrow{t_1}_{\mathcal{A}} \sigma_1 \xrightarrow{t_2}_{\mathcal{A}} \ldots \xrightarrow{t_{n-1}}_{\mathcal{A}} \sigma_{n-1} \xrightarrow{t_n}_{\mathcal{A}} \sigma_n$. For any sequence $w \in T^\star$ and any $T' \subseteq T$, $w[T']$ will denote the projection of w on elements in T'.

In the following sections, when the context is clear, the subscript $\mathcal{A}$ is omitted.

5.3 Dependencies between variables

The semantics of the AltaRica language defines the way variables are assigned a value. Flow variables are computed according to assertions, and state variables are changed when a transition is triggered. One can consider that, due to assertions, the value of a variable v depends on all other variables in the model. This is true, but some variables may be redundant. So, finding a more precise set of variables that determines the value of v can help to reduce significantly the complexity of the model. One can notice that this new set of variables is not unique: for example consider the assertion $(a = (b \wedge c)) \wedge (c \neq d)$ built with four Boolean variables. a depends on $\{b, c, d\}$; this set can be reduced to $\{b, c\}$ but also to $\{b, d\}$ because c and d are functionally dependent. The following describes one method to compute such a reduced set of variables.

Given a constraint automaton $\mathcal{A} = \langle V_S, V_F, E, T, A, I\rangle$ and $\phi \in BF(V_S \cup V_F)$, we denote by $Req_{\mathcal{A}}(\phi)$ the set of variables that influence the truth value of ϕ. The set $Req_{\mathcal{A}}(\phi)$ is built as follows:

- $Var(\phi) \subseteq Req_{\mathcal{A}}(\phi)$
- $\forall a \in A \cup I$, $Var(a) \cap Req_{\mathcal{A}}(\phi) \neq \emptyset \Rightarrow Var(a) \subseteq Req_{\mathcal{A}}(\phi)$
- $\forall t = (g, e, \alpha) \in T$, $Dom(\alpha) \cap Req_{\mathcal{A}}(\phi) \neq \emptyset \Rightarrow Var(t) \subseteq Req_{\mathcal{A}}(\phi)$ where $Var(t)$ denotes the set of variables occurring in g and α.

We have to notice that this construction depends on the writing of the assertion A. The goal of this section is to show that the correctness of the reduction algorithm does not depend on the way the assertion is written. The next section explain how to rewrite A in order to obtain better results.

The *reduction of $\mathcal{A}$ to ϕ*, denoted $\mathcal{A}^\phi$, is the restriction of $\mathcal{A}$ to variables in $Req_{\mathcal{A}}(\phi)$. The constraint automaton $\mathcal{A}^\phi = \langle V_S^\phi, V_F^\phi, E^\phi, T^\phi, A^\phi, I^\phi\rangle$ is defined as follows (t_ϵ is the macro-transition $\langle \texttt{true}, \epsilon, \emptyset\rangle$):

- $V_S^\phi = V_S \cap Req_{\mathcal{A}}(\phi)$
- $V_F^\phi = V_F \cap Req_{\mathcal{A}}(\phi)$
- $T^\phi = \{\langle g, e, \alpha\rangle \in T | Dom(\alpha) \cap Req_{\mathcal{A}}(\phi) \neq \emptyset\} \cup \{t_\epsilon\}$
- $E^\phi = \{e \in E | \exists (g, e, \alpha) \in T^\phi\}$
- $A^\phi = \{a \in A | Var(a) \cap Req_{\mathcal{A}}(\phi) \neq \emptyset\}$

- $I^\phi = \{i \in I | Var(i) \cap Req_{\mathcal{A}}(\phi) \neq \emptyset\}$

We now prove that $\mathcal{A}^\phi$ is sufficient to compute sequences of transitions that produce a configuration satisfying ϕ.

In the sequel, for any assignment $\sigma \in F(V_S \cup V_F, \mathcal{D})$ we will denote $\pi(\sigma)$ the restriction $\sigma[Req_{\mathcal{A}}(\phi)]$.

The following property states that each valuation can be decomposed into two parts; the first one satisfies assertions in A^ϕ and the second one satisfies the others.

Property 1 $\forall \sigma \in F(V_S \cup V_F, \mathcal{D}), \sigma \in [\![A]\!] \iff \sigma[Req_{\mathcal{A}}(\phi)] \in [\![A^\phi]\!] \wedge \sigma[(V_S \cup V_F) \setminus Req_{\mathcal{A}}(\phi)] \in [\![A \setminus A^\phi]\!]$

The following lemmas are illustrated on the figure 5.1.

Lemma 3 $\forall \sigma_0, \sigma_1, \sigma'_1 \in [\![A]\!], \forall t \in T, \sigma_0 \xrightarrow{t} \sigma_1 \wedge \sigma_1 \xrightarrow{t_\epsilon} \sigma'_1 \Rightarrow \sigma_0 \xrightarrow{t} \sigma'_1$

Proof: Direct consequence of AltaRica semantics. $\Diamond$

Lemma 4 $\forall \sigma_0, \sigma_1 \in [\![A]\!], \forall t \in T \setminus T^\phi, \sigma_0 \xrightarrow{t} \sigma_1 \Rightarrow \exists \sigma'_0 \in [\![A]\!], \pi(\sigma'_0) = \pi(\sigma_1) \wedge \sigma_0 \xrightarrow{t_\epsilon} \sigma'_0$

Proof: Let $t = \langle g, e, \alpha \rangle$. Since $Dom(\alpha) \cap Req_{\mathcal{A}}(\phi) = \emptyset$, t does not modify state variables belonging to $Req_{\mathcal{A}}(\phi)$; thus, $\sigma_0[V_S \cap Req_{\mathcal{A}}(\phi)] = \sigma_1[V_S \cap Req_{\mathcal{A}}(\phi)]$. The expected σ'_0 is defined by: $\sigma'_0(v) = \sigma_1(v)$ if $v \in V_F \cap Req_{\mathcal{A}}(\phi)$ and $\sigma'_0(v) = \sigma_0(v)$ for other variables. By construction we have $\pi(\sigma'_0) = \pi(\sigma_1)$. Then $\sigma'_0 \in [\![A]\!]$ because, since both σ_0 and σ_1 belong to $[\![A]\!]$, we have:

- $\sigma'_0[(V_S \cup V_F) \setminus Req_{\mathcal{A}}(\phi)] = \sigma_0[(V_S \cup V_F) \setminus Req_{\mathcal{A}}(\phi)] \in [\![A \setminus A^\phi]\!]$
- $\sigma'_0[Req_{\mathcal{A}}(\phi)] = \sigma_1[Req_{\mathcal{A}}(\phi)] \in [\![A^\phi]\!]$

Finally, since $\sigma_0[V_S] = \sigma'_0[V_S]$, the semantics gives $\sigma_0 \xrightarrow{t_\epsilon} \sigma'_0$. $\Diamond$

Lemma 5 $\forall \sigma_0, \sigma_1, \sigma'_0 \in [\![A]\!], \forall t \in T^\phi, \sigma_0 \xrightarrow{t} \sigma_1 \wedge \pi(\sigma_0) = \pi(\sigma'_0) \Rightarrow \exists \sigma'_1 \in [\![A]\!], \sigma'_0 \xrightarrow{t} \sigma'_1 \wedge \pi(\sigma_1) = \pi(\sigma'_1)$

Proof: We assume hypotheses for some transition $t = \langle g, e, \alpha \rangle$. Now, consider the valuation σ'_1 such that σ'_1 is equal to σ_1 for all variables in $Req_{\mathcal{A}}(\phi)$ and is equal to σ'_0 elsewhere. By construction, $\sigma'_1[Req_{\mathcal{A}}(\phi)] \in [\![A^\phi]\!]$ and $\sigma'_1[(V_S \cup V_F) \setminus Req_{\mathcal{A}}(\phi)] \in [\![A \setminus A^\phi]\!]$, thus, due to property 1, σ'_1 belongs to $[\![A]\!]$. Since $\pi(\sigma_0) = \pi(\sigma'_0)$, we have $\sigma'_0 \in [\![g]\!]$.

Now, let $v \in Dom(\alpha)$. Since $t \in T^\phi$ we have $v \in Req_{\mathcal{A}}(\phi)$ and thus, $\sigma'_1(v) = \sigma_1(v) = [\![\alpha(v)]\!](\sigma_0) = [\![\alpha(v)]\!](\sigma'_0)$. In the case where $v \in V_S \setminus Dom(\alpha)$, $\sigma'_1(v) = \sigma'_0(v)$. We can conclude that $\sigma'_0 \xrightarrow{t} \sigma'_1$. $\Diamond$

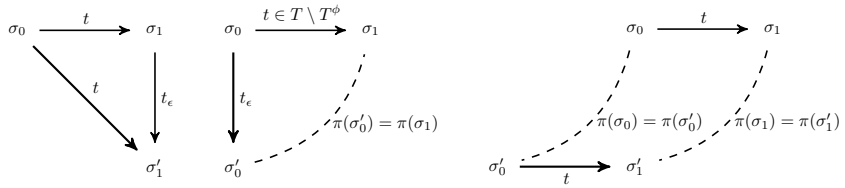


Figure 5.1: Illustration of the lemmas

Corollary 2 *Let $n > 1$, $\sigma_0 \in [\![I]\!] \cap [\![A]\!]$ and for $i = 1, \ldots, n$, $\sigma_i \in [\![A]\!]$ and $t_i \in T$, such that $\sigma_{i-1} \xrightarrow{t_i} \sigma_i$. If $j \in [1, n]$ is such that $t_j \in T \setminus T^\phi$ and for all $j < k \leq n$, $t_k \in T^\phi$ then there exist $\sigma'_i \in [\![A]\!]$ for $i = j, \ldots, n$ such that:*

- $\sigma'_i \xrightarrow{t_{i+1}} \sigma'_{i+1}$ *for* $i = j, \ldots, n-1$
- $\pi(\sigma_i) = \pi(\sigma'_i)$ *for* $i = j, \ldots, n$
- $\sigma_{j-2} \xrightarrow{t_{j-1}} \sigma'_j$ *if* $j > 1$ *or* $\sigma'_j \in [\![I]\!]$ *if* $j = 1$

Proof: In the case where $j = n$, lemmas 3 and 4 are sufficient. For cases where $j > 1$, after applying 3 and 4 to remove t_j, lemma 5 is used to built the sequence from σ'_j to σ'_n. For the case where t_j is the first macro-transition of the sequence (i.e $j = 1$), lemma 4 states that $\sigma_0 \xrightarrow{t_\epsilon} \sigma'_1$. Due to AltaRica semantics, since σ_0 is an initial configuration, any configuration reachable by t_ϵ is also an initial configuration. ◊

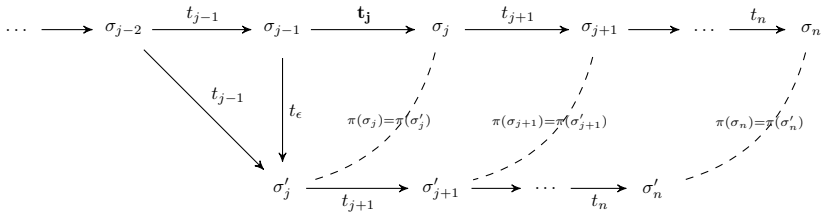


Figure 5.2: Illustration of corollary 2.

Previous corollary is illustrated on figure 5.2. The following theorem, that is a consequence of corollary 2, states that only sequences of transitions belonging to T^ϕ are actually relevant to reach configuration satisfying ϕ.

Theorem 4 $\forall \sigma_0 \in [\![I]\!]$, $w \in T^\star$, $\sigma \in [\![\phi]\!]$, $\sigma_0 \xrightarrow{w} \sigma \Rightarrow \exists \sigma'_0 \in [\![I]\!]$, $\sigma' \in [\![\phi]\!]$, $\sigma'_0 \xrightarrow{w[T^\phi]} \sigma'$.

Proof: Corollary 2 permits to remove successively from w macro-transitions that do not belong to T^ϕ without loosing the reachability of $\pi(\sigma_n)$. $\Diamond$

Property 1 and theorem 4 allow us to remove parts of $\mathcal{A}$ that are not related to $Req_{\mathcal{A}}(\phi)$ without loosing relevant sequences yielding configurations in $[\![\phi]\!]$.

Does $\mathcal{A}^\phi$ produces sequences that are not sequences of $\mathcal{A}$? As we will show below, this can not occur if $[\![I]\!] \cap [\![A]\!] \neq \emptyset$ i.e $\mathcal{A}$ has at least one initial configuration.

Lemma 6 $\forall \sigma_1, \sigma_2 \in [\![A^\phi]\!]$, $\forall t \in T^\phi$, $\sigma_1 \xrightarrow{t}_{\mathcal{A}^\phi} \sigma_2 \Rightarrow \forall \sigma'_1 \in [\![A]\!] \cap \pi^{-1}(\sigma_1)$, $\exists \sigma'_2 \in [\![A]\!] \cap \pi^{-1}(\sigma_2)$, $\sigma'_1 \xrightarrow{t}_{\mathcal{A}} \sigma'_2$.

Proof: If σ'_1 exists it suffices to define σ'_2 as follows: for all $v \in Req_{\mathcal{A}}(\phi)$, $\sigma'_2(v) = \sigma_2(v)$ and if $v \in (V_S \cup V_F) \setminus Req_{\mathcal{A}}(\phi)$, $\sigma'_2[v] = \sigma'_1(v)$. By construction σ'_2 belongs to $\pi^{-1}(\sigma_2)$. $\sigma'_2[Req_{\mathcal{A}}(\phi)] = \sigma_2 \in [\![A^\phi]\!]$ and $\sigma'_2[(V_S \cup V_F) \setminus Req_{\mathcal{A}}(\phi)] = \sigma'_1[(V_S \cup V_F) \setminus Req_{\mathcal{A}}(\phi)] \in [\![A \setminus A^\phi]\!]$ thus $\sigma'_2 \in [\![A]\!]$. Since t does not modify variables out of $Req_{\mathcal{A}}(\phi)$, $\sigma'_1 \xrightarrow{t}_{\mathcal{A}} \sigma'_2$. $\Diamond$

By induction one can easily prove that previous lemma can be generalized to sequences in $(T^\phi)^\star$. This means that if a sequence $w \in (T^\phi)^\star$ can be triggered from a configuration σ_0 of $\mathcal{A}^\phi$, the non-emptiness of $\pi^{-1}(\sigma_0) \cap [\![A]\!]$ ensures that w can also be executed in $\mathcal{A}$ from some configuration in $\pi^{-1}(\sigma_0)$.

Theorem 5 $[\![I]\!] \cap [\![A]\!] \neq \emptyset \Rightarrow \forall \sigma_0, \sigma \in [\![A^\phi]\!], \forall w \in (T^\phi)^\star, (\sigma_0 \in [\![I^\phi]\!] \wedge \sigma_0 \xrightarrow{w}_{\mathcal{A}^\phi} \sigma)$ $\Rightarrow (\exists \sigma'_0, \sigma' \in [\![A]\!], \sigma'_0 \in [\![I]\!] \wedge \sigma'_0 \xrightarrow{w}_{\mathcal{A}} \sigma' \wedge \pi(\sigma'_0) = \sigma_0 \wedge \pi(\sigma') = \sigma)$

Proof: Since $\sigma_0 \in [\![I^\phi]\!] \cap [\![A^\phi]\!]$ and $[\![I]\!] \cap [\![A]\!] \neq \emptyset$, there exists $\sigma'_0 \in [\![I]\!] \cap [\![A]\!]$ such that $\pi(\sigma'_0) = \sigma_0$. Then generalization to sequences of lemma 6 permits to conclude that w can be triggered from σ'_0 yielding a configuration σ' belonging to $\pi^{-1}(\sigma) \cap [\![A]\!]$. $\Diamond$

5.4 Reduction using functional dependencies

In many cases, all variables of a constraint automaton are mutually dependent and no gain is obtained with the reduction because $Req_{\mathcal{A}}(\phi) = V_S \cup V_F$. However we can obtain a better result if the system can be actually partitioned into disconnected parts.

The mutual dependency between variables is due to the semantics of the AltaRica language that wires variables of components using assertions. From a formal point of view, assertions define a relation between variables without any notion of orientation or causality between variables. However, users often thought assertions as an assignment or, at least, a data flow from one point to another. As a consequence, many models possess an implicit orientation given by the way engineers describe systems and, often, many flow variables functionally depend on other variables. These functional dependencies can be used

to remove some assertions and thus reduce connectivity between components of the automaton.

Section 5.5 gives an algorithm that build two sets C and FD such that:

- $C \subseteq BF(V_S \cup V_F)$, $FD \subseteq V_F \times T(V_S \cup V_F)$
- $C \cup \{v = t \in BF(V_S \cup V_F) \mid (v,t) \in FD\}$ and the original set of assertions A have the same semantics.
- $\forall (v,t),(v',t') \in FD, v = v' \Rightarrow t = t'$ i.e each flow variable appears at most once on the left hand size of FD.

Each of couple $(v,f) \in FD$ associates to a flow variable a term f that defines the value of v. Using these *functional dependencies*, one can reduce further the constraint automaton prior the reduction given in the previous section. Actually, each flow variable appearing on the first component of a couple in FD can be replaced directly by its associated function. This variable can then be suppressed from the model. This process has the advantages of reducing the number of flow variables and remove some dependencies. Of course, the same substitution has to be applied to the input formulas.

5.5 Computation of functional dependencies

In this section we give the algorithm used to compute functional dependencies between variables of a constraint automaton. If A is the set of assertions of the considered automaton, the algorithm creates two new sets:

1. The first one is a set of functional dependencies $FD \subseteq V_F \times T(V_S \cup V_F)$ that associates to a flow variable v a term t built over $(V_S \cup V_F) \setminus \{v\}$. The equation $v = t$ is an invariant of the model that is semantically equivalent to an assertion belonging to A.
2. The second one, $C \subseteq BF(V_S \cup V_F)$ is simply assertions for which no functional dependency has been found.

The main algorithm (`compute-functional-dependencies` below) is quite simple. For each assertion c it looks for a functional dependency that is semantically equivalent (line 4) i.e a variable $v \in Var(c)$ and a term $f \in T(Var(c) \setminus \{v\})$ such that formulae c and $v = f$ have the same solutions. If such a couple is not obtained the assertion is kept as is and stored into C (lines 5-6). If a dependency is found it is added to FD. In order to prevent the creation of cycles, we maintain DEP a dependency relation between variables (lines 8-11); if $(v,v') \in DEP$ then v depends on v'.

Algorithm 4 `compute-functional-dependencies (`A`)`

```
1:  FD ← ∅, C ← ∅
2:  DEP ← ∅
3:  for all c ∈ A do
4:    (v, f) ← look-for-fd (c, DEP)
5:    if (v, f) = (⊥, ⊥) then
6:      C ← C ∪ {c}
7:    else
8:      FD ← FD ∪ {(v, f)}
9:      for all v' ∈ Var(f) do
10:       DEP ← DEP ∪ {(v, v')}
11:     end for
12:   end if
13: end for
14: return ⟨FD, C⟩
```

Algorithm `look-for-fd` (page 48) checks if there exists a functional dependency in the semantics of an assertion c. The algorithm first simply checks if the assertion c is an equality between a flow variable x and another term t (lines 2-3). If this is the case and if variables appearing in t do not depend on x (line 4) then the couple (x, t) is returned.

If no functional dependency is found syntactically, the search is realized on the semantics of c (lines 15-21). For each flow variable x appearing in c that does not depend (w.r.t DEP) on other variables in c, we check if the value of x is a function of other variables. This test can be realized by a solver.

Line 18, `build-funct (`x`, `c`)` built the term $t \in T(Var(c) \setminus \{x\})$ such that $x = t$ and c are semantically equivalent.

5.6 Example

In this section we consider a system composed of 4 identical leaf nodes. Such nodes simply transmit their "input" to their "output" unless a failure occurs in which case the output is set to `false`. Nodes are organized into two columns. On the right column, nodes $C01$ and $C11$ receive as input the conjunction of the output of nodes in the left column ($C00$ and $C10$). The system is depicted on the figure 5.3 and below the figure, quantitative data about the top-level node (`System`) are given.

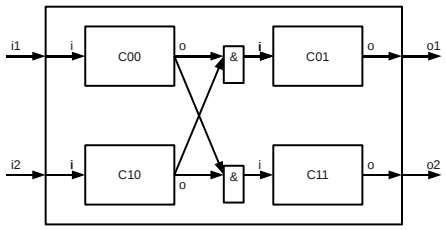
Algorithm 5 `look-for-fd (`c`, `DEP`)`

1: /* syntactic test */
2: **if** c has the form $T_1 = T_2$ **then**
3: **for all** $x \in \{T_1, T_2\} \cap V_F$ **do**
4: **if** $\forall v \in Var(c) \setminus \{x\}, (v, x) \notin DEP^\star$ **then**
5: **if** $x = T_1$ **then**
6: **return** (T_1, T_2)
7: **else**
8: **return** (T_2, T_1)
9: **end if**
10: **end if**
11: **end for**
12: **end if**
13:
14: /* semantic test */
15: **for all** $x \in Var(c) \cap V_F$ **do**
16: **if** $\forall v \in Var(c) \setminus \{x\}, (v, x) \notin DEP^\star$ **then**
17: **if** $\forall (d_1, d_2) \in \mathcal{D}^2, d_1 \neq d_2 \Rightarrow [\![c[x/d_1]]\!] \cap [\![c[x/d_2]]\!] = \emptyset$ **then**
18: **return** $(x,$`build-funct (`x`, `c`)`$)$
19: **end if**
20: **end if**
21: **end for**
22: **return** $(\bot, \bot)$

```
node C
  flow i, o : bool
  state s : { ok, ko };
  init s := ok;
  event fail;
  trans
   s = ok |- fail -> s := ko;
  assert
    if (s = ok)
      then (o = i)
      else (o = false);
edon

node Equipment
  flow i1, i2, o1, o2 : bool;
  sub C00, C01, C10, C11 : C;
  assert
    i1 = C00.i;
    i2 = C10.i;
    C01.i = (C00.o & C10.o);
    C11.i = (C00.o & C10.o);
    C01.o = o1;
    C11.o = o2;
edon

node System
  sub E : Equipment;
edon
```




```
=== System ===
statistics:
 number of variables : 16
   flow variables : 12
   state variables : 4
 max cardinality : 2
 number of events : 5
 number of functional dependencies : 10
 number of constraints : 0
 number of transitions : 5
```

Figure 5.3:

Now, assume we want to observe the output variable `o1`. The "reduced" system generated by the projection described above is given below. One can remark that this latter does not contain anymore components related to component `C11`. The dependency graphs for this system, before and after the projection, are depicted on figures 5.5 (page 51) and 5.6 (page 52).

```
// statistics:
//  number of variables : 5
//    flow variables : 2
//    state variables : 3
//  max cardinality : 2
//  number of events : 4
//  number of functional dependencies : 0
//  number of constraints : 0
//  number of transitions : 4
node System
 flow // 2 flow variables
   'E.C00.i' : bool;
   'E.C10.i' : bool;
 state // 3 state variables
   'E.C00.s' : { ok, ko };
   'E.C01.s' : { ok, ko };
   'E.C10.s' : { ok, ko };
 init
   'E.C10.s' := ok,
   'E.C01.s' := ok,
   'E.C00.s' := ok;
 event // 4 events
   'E.C00.fail';
   'E.C01.fail';
   'E.C10.fail';
 trans
   ('E.C10.s' = ok) |- 'E.C10.fail' -> 'E.C10.s' := ko;
   ('E.C01.s' = ok) |- 'E.C01.fail' -> 'E.C01.s' := ko;
   ('E.C00.s' = ok) |- 'E.C00.fail' -> 'E.C00.s' := ko;
edon
```

Projection of `System` node for the observation of `E.o1`.

Now we create a loop into the system of the previous example. The output of the system `o2` is redirected into the input `i1`. The description of the system is exactly the same as previously except that the `System` node contains the assertion describing the loop (see figure 5.4).

```
node System
  sub E : Equipment;
  assert E.o2 = E.i1;
edon
```

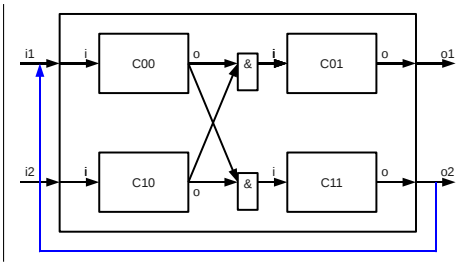


Figure 5.4: A feedback loop is added that wires up output `o2` to input `i1`

If we keep the observation of the output variable `o1` as our objective, the reduced system generated by the algorithm is "almost" the whole system. In

fact, even if applying functional dependencies to the model reduce the number of variables, all behaviours and all actually usefull variables are kept.

```
// statistics:
//  number of variables : 6
//    flow variables : 2
//    state variables : 4
//  max cardinality : 2
//  number of events : 5
//  number of functional dependencies : 0
//  number of constraints : 1
//  number of transitions : 5
node System
 flow // 2 flow variables
   'E.C00.i' : bool;
   'E.C10.i' : bool;
 state // 4 state variables
   'E.C00.s' : { ok, ko };
   'E.C01.s' : { ok, ko };
   'E.C10.s' : { ok, ko };
   'E.C11.s' : { ok, ko };
 init
   'E.C11.s' := ok,
   'E.C10.s' := ok,
   'E.C01.s' := ok,
   'E.C00.s' := ok;
 event // 5 events
   'E.C00.fail';
   'E.C01.fail';
   'E.C10.fail';
   'E.C11.fail';
 assert // 1 actual constraints
   ((('E.C11.s' = ok) and ((('E.C00.s' = ok) and 'E.C00.i') and (('E.C10.s' = ok)
       and 'E.C10.i'))) = 'E.C00.i');
 trans
   ('E.C11.s' = ok) |- 'E.C11.fail' -> 'E.C11.s' := ko;
   ('E.C10.s' = ok) |- 'E.C10.fail' -> 'E.C10.s' := ko;
   ('E.C01.s' = ok) |- 'E.C01.fail' -> 'E.C01.s' := ko;
   ('E.C00.s' = ok) |- 'E.C00.fail' -> 'E.C00.s' := ko;
edon
```

The dependency graphs for this new system, before and after the projection, are depicted on figures 5.7 (page 53) and 5.8 (page 54).

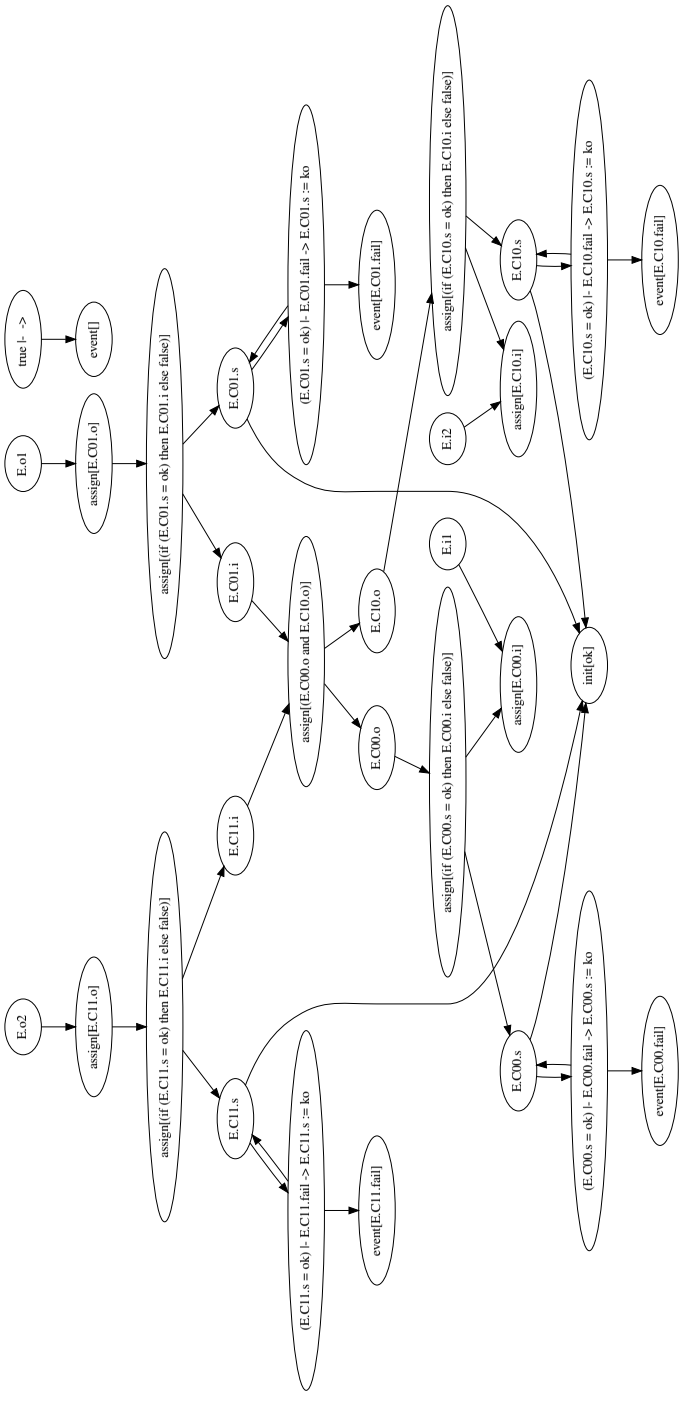


Figure 5.5: Dependency graph of the acyclic system.

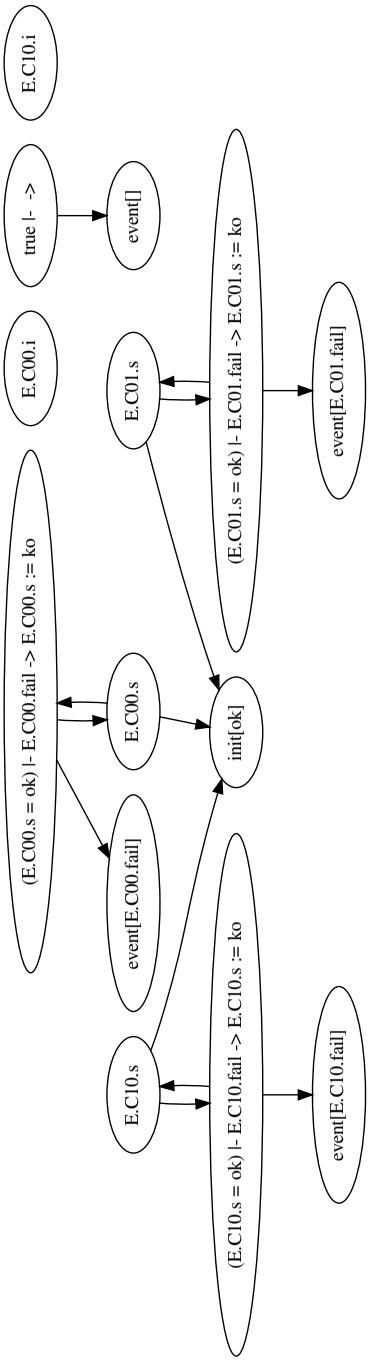


Figure 5.6: Dependency graph of the acyclic system after the projection.

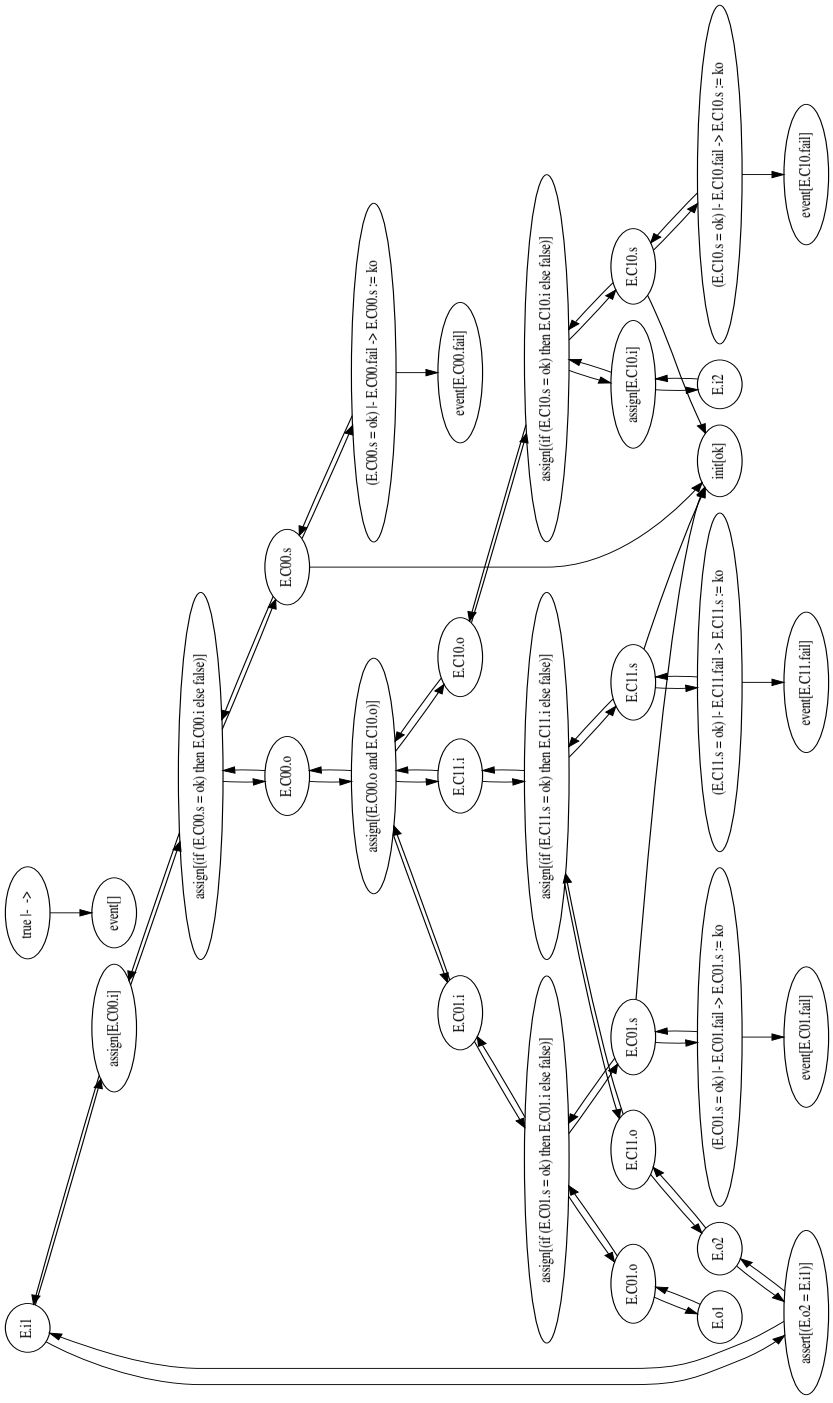


Figure 5.7: Dependency graph of the system with a loop.

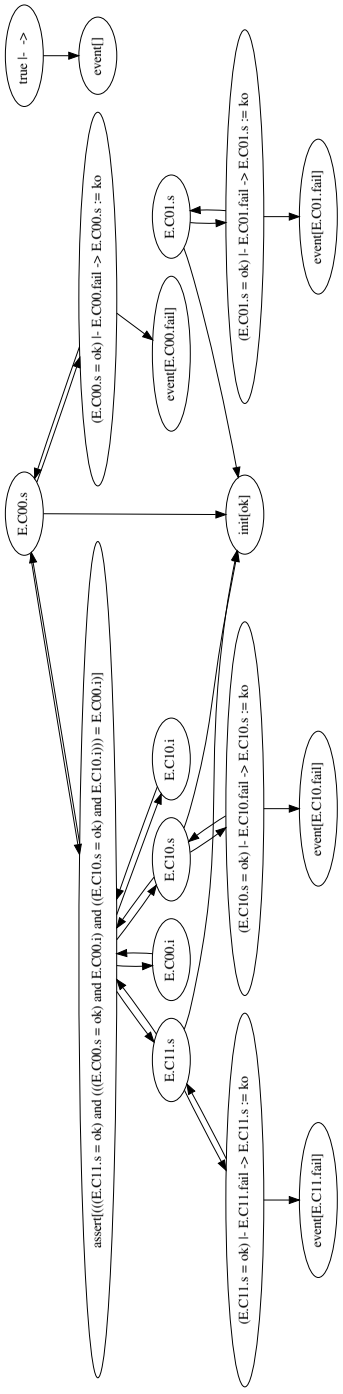


Figure 5.8: Dependency graph of the system with a loop after the projection.

Chapter 6

Experiments

6.1 The `cuts` command

Computations of sequences, cuts and minimization are accessible in ARC using the `cuts` command using the following syntax:

`cuts [`*`options`*`] `*`nodeid acceptance-condition`*

where options are:

`--visible-tags=`vt_1`,...,`vt_n

`--disabled-tags=`dt_1`,...,`dt_m

`--min`

`--ordered=`k

This command[1] permits to compute sets of scenarios that lead the model described by $nodeid$ into a configuration satisfying the formula $acceptance - condition$. The formula $acceptance - condition$ can be any Boolean formula over all variables of $nodeid$.

The command returns a Boolean formula that encodes cuts yielding the expected configurations. The formula is given using the syntax of the ARALIA tool[8]. Literals that compose cuts are elementary events of the model that have been specified as *visible* by the user. By default, the set of visible events is empty; thus, the result of the command is 0 (`false`) or 1 (`true`) depending on the existence or not expected configurations.

The following session shows an example of cuts computation. The studied node is a simple counter from 0 to 10. The counter can be incremented by one unit using `inc` event or by two units using `inc2` event. The first event is labelled with event `attr1` and the second one with `attr2`. As explained above the result given by `cuts` is a Boolean constant (here 1 because expected states are reachable from initial state).

[1] Paragraphs of this section have been extracted from ARC handbook.

```
$ cat cuts-example.alt
node Counter
  state count : [0,10]; init count := 0;
  event inc : attr1;
        inc2 : attr2;
  trans true |- inc -> count := count + 1;
        true |- inc2 -> count := count + 2;
edon

$ arc -qb cuts-example.alt -c 'cuts Counter "count>=3"'
N0 := 1;
root := N0;
```

If we request ARC to compute scenarios but considering that `inc` and `inc2` must appear in the result we obtain a Boolean formula that encodes two sequences:

```
$ arc -qb cuts-example.alt -c 'cuts --visible-tags=attr1,attr2 \
  Counter "count>=3"'
N2 := 1;
N1 := (`inc2' ? -N2 : N2);
N0 := (`inc' ? -N2 : N1);
root := -N0;
```

Now if only event `inc2` is observed we obtain yet the Boolean constant 1.

```
$ arc -qb cuts-example.alt -c 'cuts --visible-tags=attr2 \
  Counter "count>=3"'
N0 := 1;
root := N0;
```

Why do we obtain 1? Because, event `inc2` is implicitly simplified. Actually, non-observed events are replaced by 1 in the formula obtained when all events are observed. In our example the formula is `inc` or `inc2`; thus when `inc` is assigned the constant 1 the formula is simplified into 1.

The option `--disabled-tags=`dt_1`,...` permits to indicate that events labelled with one of the tags dt_is are forbidden in scenarios:

```
$ arc -qb cuts-example.alt -c 'cuts --visible-tags=attr2 \
   --disabled-tags=attr1 Counter "count>=3"'
N1 := 1;
N0 := (`inc2' ? -N1 : N1);
root := -N0;
```

By default, the command computes sets of events. The option `--ordered=`k indicates that the command has to compute ordered sequences of events and moreover, each sequence can not contain more than k visible events. If we come back on previous example and limit the number of visible events to 3. Note that ordered sequences are not given as Boolean formulas.

```
$ arc -qb cuts-example.alt -c 'cuts --visible-tags=attr1,attr2 \
   --ordered=3 Counter "count>=3"'
(inc2, inc2)
(inc2, inc)
(inc, inc2)
(inc, inc, inc2)
(inc, inc, inc)
```

Finally the option `--min` can be used to filter sets to minimal elements; it can be used for both cuts and sequences. If cuts are computed the minimality criterion is the inclusion. In the case of ordered sequences, sub-sequences are considered.

```
$ arc -qb cuts-example.alt -c 'cuts --visible-tags=attr1,attr2 \
  --ordered=3 --min Counter "count>=3"'
(inc2, inc2)
(inc2, inc)
(inc, inc2)
(inc, inc, inc)
```

6.2 Experimental results

6.2.1 Industrial model

Following tables summarize results of experiments on an industrial model. Table 6.1 gives sizes of the model in terms of its number of variables and events. The number of objectives is number of formulas specifying unwanted configurations. The table also gives execution times of ARC for the 2388 computations of minimal cuts and sequences up to order 3. Experiments were conducted on a standard laptop computer (Intel Centrino 2 - 2.53GHz) with processes limited to 2.5 Gb of memory. Times are expressed in CPU time.

# state	**# flow**	**# events**	**# objectives**	**Mincuts**	**Sequences ($k \leq 3$)**
1349	7713	2268	2388	1775	4989

Table 6.1: Sizes of the model and computation times in CPU seconds

We can consider that each formula that specify a set of unexpected configurations is treated in less than one second; however there exists an incompressible amount of time due to the computation of the flat semantics of the model. This preprocessing time takes around 20 seconds for this model.

6.2.2 Model-checking models

We have experimented proposed algorithms on models stemmed from the model-checking community plus another model mainly used for benchmark-

ing purpose. Table 6.2 gathers results of these experiments; the first column gives the name of the model and its parameter(s).

Model name	Reachables	Mincuts	Ordered with $k \leq 3$	# variables		# transitions		# constraints	
				before	after	before	after	before	after
burns 2x2	5729	0.030	0.025	11	11	41	41	0	0
burns 3x1	236088	0.084	0.098	16	16	82	82	0	0
burns 3x2	1032183	0.148	0.106	16	16	82	82	0	0
burns 3x3	2764800	0.207	0.107	16	16	82	82	0	0
lamport 2	1569	0.048	0.026	11	11	53	43	0	0
lamport 3	257704	2.941	0.114	16	16	139	103	0	0
lamport 4	75599953	54.656	0.421	21	21	289	201	0	0
lift 5	14336	1.869	0.068	25	18	25	25	0	0
lift 6	71680	7.776	0.122	29	21	29	29	0	0
lift 7	344064	28.354	0.252	33	24	33	33	0	0
lift 8	1605632	107.308	0.436	37	27	37	37	0	0
lift 9	7340032	aborted	0.863	41	30	41	41	0	0
peterson 2	20	0.004	0.003	14	5	9	9	2	0
peterson 3	417	0.058	0.011	28	8	19	19	3	0
peterson 4	9272	4.294	0.040	46	11	33	33	4	0
peterson 5	223105	145.792	0.102	68	14	51	51	5	0
stress 2	5	0.001	0.000	6	3	3	3	1	1
stress 4	17	0.003	0.004	12	5	5	5	1	1
stress 8	257	0.006	0.013	24	9	9	9	1	1
stress 16	65537	0.015	0.296	48	17	17	17	1	1
stress 32	4294967297	0.055	15.178	96	33	33	33	1	1

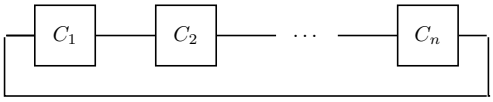
Table 6.2: Results of experiments on classic examples from model-checking domain. *reachables* column gives the number of reachable configurations of the reduced model. *mincuts* and *ordered with* columns gives execution times for the computation of, respectively, minimal cuts and minimal sequences. The last six columns compare sizes (i.e number of variables, transitions and constraints) of models with their reduced version.

Mutual exclusion models: `Burns`, `Lamport` and `Peterson` are well-known mutual-exclusion algorithms used to permit read/write accesses to a resource shared by N processes. Even if models are quite small in terms of number of variables, transitions and assertions, their semantics grow exponentially with respect to the number of processes.

In the context of formal methods, these models are checked against several kind of behaviourial properties: absence of deadlocks and livelocks, fairness and others, including mutual exclusion. The latter property belongs to the family of safety properties and can be treated by our algorithms. Thus, in our experiment we have specified as unexpected configurations, those for which mutual exclusion is violated.

Lift model: This model is inspired from a system described in [10]. As is indicated by its name, the model describes a lift and its environment. The model is parameterized by the number of floors. Model-checking methods are used to check that movements of the cage are correct e.g. each request at a floor will be satisfied. The safety property used to check our algorithms is the presence or not of the cage at a floor where doors are opened.

Stress test model: This last model is a simple circular-pipeline composed of $n \geq 4$ components connected in series with the output of last component connected to the input of the first one (see figure below). The unexpected event is the failure of four components fixed in advance. Failures do not depend on flows; thus, one could remove all flow variables of the model and keep only the four observed components to study unexpected states. However, since there exists a loop relating variables, the reduction algorithm fails to simplify the model.



The results given in table 6.2 deserve some comments. First of all, except for `stress` case, models describe functional behaviours and even algorithms (for mutual exclusion models); thus, no failure event exists. In order to evaluate algorithms we had to declare all events as visible ones; otherwise computation would have been equivalent to simple reachability analysis of unexpected states.

Second, yet excepting `stress` case, modeled systems are not faulty which means that unexpected states are not reachables. Thus, only `stress` cases have non empty sets of cuts but results can be easily checked.

Except for `lift` with more thant 9 floors, all test cases terminate in few minutes for both mincuts and sequences. Note that comparing mincuts and sequences computation times has no sense. Actually, since all events are visible, the algorithm that computes sequences terminates research of unexpected states after 3 transitions while the one that computes cuts builds the whole set of reachable configurations.

Efficiency of reduction algorithm can be evaluated with the last six columns. One can easily notice that the technique is ineffective because all transitions are preserved. Results for `lamport` model are misleading; indeed, several macro-transitions are deliberately made disabled (with `false` guards). Several constraints and variables are removed but this is due to discovery of functional dependencies and not to reduction algorithm (e.g. `lift` models). Only `stress` test case has an actual constraint (due to the loop); other models define only functional dependencies between state and flow variables. The inefficiency of reduction is not so surprising because unexpected configurations involve the whole system.

Chapter 7

Conclusion

In this report we have described algorithms implemented into ARC to compute scenarios that lead an AltaRica model into unwanted configurations. This process, depicted on figure 7.1 below, can be summarized as follows:

1. As usual the AltaRica model $\mathcal{M}$ is translated into a unique constraint automaton $\mathcal{A}$ that describes all behaviours of the system.
2. A preprocessing step is applied to $\mathcal{A}$ in order to remove parts of the model that are not related to configurations specified by the Boolean formula ϕ (that models unwanted states). This gives a new constraint automaton $\mathcal{A}_\phi$.
3. Then $\mathcal{A}_\phi$ is augmented with an observer according to the specified set of visible events V. For the augmented automaton we symbolically compute, using DDs, reachable configurations that intersect ϕ.
4. Finally the DD that encodes scenarios is translated into a RLDD that encodes minimal ones.

We have experimented our algorithms with a large model taken from the industry and with model-checking test-cases. Results are promising but models were not good representatives of those that can be found for instance in the safety assessment domain.

Actually model-checking test-cases are not good candidates to evaluate such algorithms because models essentially describe systems with no bugs and were elementary failures does not exist (models represent algorithms). Second, the large industrial model was not checked against a formula specifying a global failure but rather small and localized abnormal configurations; this explain why reduction algorithm (see chapter 5) worked fine.

Future works should aim at determining how formulas that specify unwanted configurations can be decomposed in such a way that reduction algorithm can be applied to each sub-formulas and recomposed afterwards.

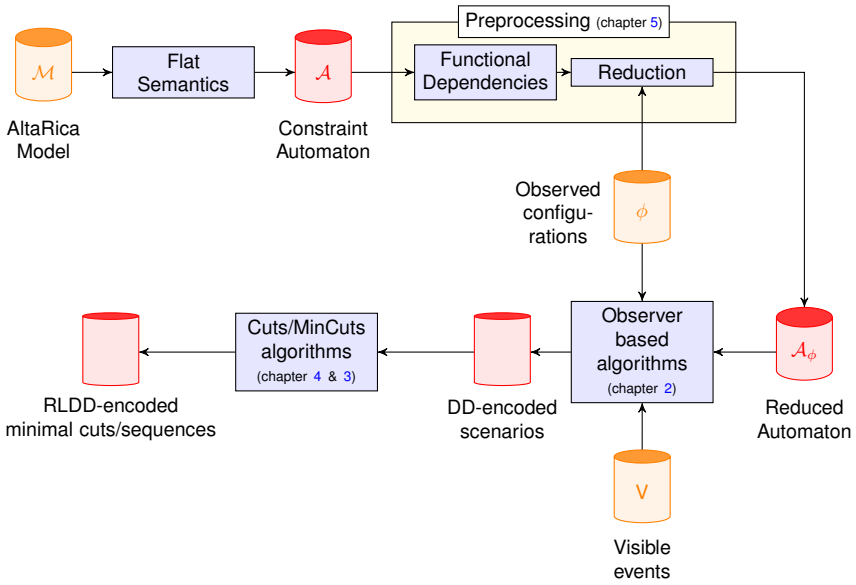


Figure 7.1: Pipeline used to compute sets of failure scenarios for an AltaRica model.

Another important task will be to check our algorithms on more representative test-cases from the industry.

Bibliography

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

[2] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[3] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[4] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *Logic in Computer Science, 1989. LICS '89, Proceedings., Fourth Annual Symposium on*, pages 353 –362, jun 1989.

[5] M.-M. Corsini and A. Rauzy. Toupie: The μ-calculus over finite domains as a constraint language. *J. Autom. Reasoning*, 19(2):143–171, 1997.

[6] Johan de Kleer, Alan K. Mackworth, and Raymond Reiter. Characterizing diagnoses. In *AAAI'90: Proceedings of the eighth National conference on Artificial intelligence*, pages 324–330. AAAI Press, 1990.

[7] S. Denzumi, R. Yoshinaka, S.-I. Minato, and H. Arimura. Efficient algorithms on sequence binary decision diagrams for manipulating sets of strings. Technical report, Hokkaido University, April 2011.

[8] Y. Dutuit and A. Rauzy. Exact and truncated computations of prime implicants of coherent and non-coherent fault trees within aralia. *Reliability Engineering and System Safety*, (58):127–144, 1997.

[9] A. Griffault, G. Point, and A. Vincent. *AltaRica Checker Handbook, a user-guide to ARC version 1.3*. LaBRI/MV/MF, Talence, January 2010.

[10] F. Laroussinie. *Logique temporelle avec passé pour la spécification et la vérification des systèmes réactifs*. Thèse de doctorat, Institut National Polytechnique de Grenoble, France, November 1994.

[11] E .Loekito, J. Bailey, and J. Pei. A binary decision diagram based approach for mining frequent subsequences. *Knowl. Inf. Syst.*, 24(2):235–268, 2010.

[12] S. Minato. *Binary Decision Diagrams and Applications for VLSI CAD*. Kluwer Academic Publishers, 1996.

[13] G. Point. *AltaRica : Contribution à l'unification des méthodes formelles et de la sûreté de fonctionnement*. PhD thesis, Université Sciences et Technologies - Bordeaux I, 01 2000. 2198.

[14] G. Point and A. Rauzy. Altarica - constraint automata as a description language. *European Journal on Automation*, 1999. Special issue on the *Modelling of Reactive Systems*.

[15] P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81 –98, jan 1989.

[16] A. Rauzy. Mathematical foundation of minimal cutsets. *IEEE Transactions on Reliability*, 50(4):389–396, 2001.

[17] A. Rauzy. Modes automata and their compilation into fault trees. *Reliability Engineering and System Safety*, (78):1–12, 2002.

[18] R Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.

[19] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D.C. Teneketzis. Failure diagnosis using discrete-event models. *IEEE Transactions on Control Systems Technology*, 4(2):105–124, March 1996.

[20] Stefan Schwoon. A note on on-the-fly verification algorithms. In *In Proc. of TACAS'05, LNCS*, pages 174–190. Springer-Verlag, 2005.