AltaRica 3.0
Language Specification

Michel BATTEUX Tatiana PROSVIRNOVA Antoine RAUZY

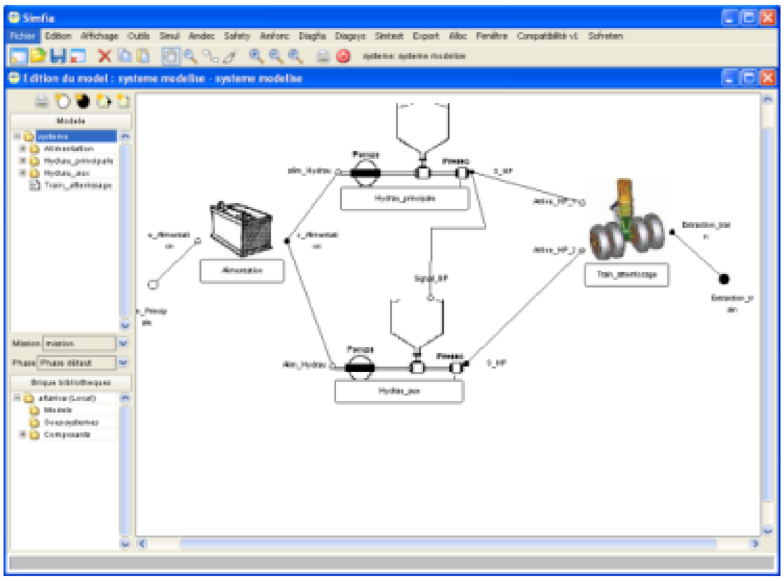
Abstract: This document defines the AltaRica 3.0 language. AltaRica 3.0 is a freely available, high level language for event driven modeling of complex systems. It is especially well suited for safety analyses and performance analyses. Models in AltaRica 3.0 are mathematically described by Guarded Transition Systems. Several assessment tools are available such as stepwise simulators, compilers to Fault Trees, generators of critical sequences, compilers to Markov chains, stochastic simulators and model-checkers

Version	Description
1.0	Initial version
1.1	Clause ’embeds’: require an alias. Clause ’clones’: added to the specification. Path identifiers: add the notions of ’main’ and ’owner’. Hiding events: change from the clause hide to the attribute ’hidden’. Clause ’constant’: defined like a parameter. Add virtual operations. Grammars of GTS: added.
1.2	Remove virtual operations (not yet available for tools).

Preface

The AltaRica project started at the end of the nineties. AltaRica is a high level modeling language dedicated to safety analyses. The main motivation for the creation of such a language was that models designed with “classical” formalisms such as Fault Trees, Markov chains, Petri nets and the like are too distant from systems under study. As a consequence, these models are hard to design, and even harder to share amongst the stakeholders and to maintain throughout the life cycle of systems. The idea was therefore to design models that would reflect better the functional and physical architecture of the systems at hand.

AltaRica has been designed to preserve all good characteristics of above cited formalisms and to have some additional expected ones. First of all, AltaRica is event centric. The primary objective of Safety and Reliability studies is actually to detect and to quantify the most probable sequences of events (failures) leading the system from a nominal state to a degraded state (accident). The semantics of AltaRica is formally defined in terms of Guarded Transitions Systems: the state of the system is described by means of variables. The system changes of state when, and only when, an event occurs. The occurrence of an event updates the value of variables. No assumption is made on the physical meaning of states and events. AltaRica is therefore a formal and agnostic language. Events can be associated with deterministic or stochastic delays so to obtain (stochastic) timed models. Guarded Transition Systems can be encapsulated into “boxes”. An AltaRica model is actually a hierarchy of “boxes” linked with “wires” that represent constraints between encapsulated variables. Encapsulated events can also be constrained by synchronizing them. From a user perspective, AltaRica models can be graphically created, edited and simulated, as illustrated below.



Three main versions of AltaRica have been designed so far. These three versions differ essentially with two respects: the way variables are updated after each transition firing and the way hierarchies of components are managed. In the first version, designed at the Computer Science Laboratory of University of Bordeaux (LaBRI), variables were updated by solving constraint systems. This model, although very powerful, was too resource consuming for industrial scale applications. A radical turn was therefore taken with the design of a data-flow version of the language. In AltaRica Data-flow, variables are updated by propagating values in a fixed order. This order is determined at compile time. At the time this document is written, most of AltaRica tools rely on this data-flow version. Experience showed that AltaRica Data-Flow can be improved in several ways, hence justifying the third version of the language: AltaRica 3.0 specified in the present document.

AltaRica 3.0 still relies on propagation based variable update, but the order in which this propagation takes place in determined at execution time. Technically, a fixpoint is computed. This new update scheme does not increase much computation times but extends significantly the expressivity of the language. It makes it possible to handle looped systems, i.e. systems in which variables depend of each other in a circular way. Another new feature of AltaRica 3.0 is the notion of guarded synchronization that both increases the expressivity and unifies transitions and synchronizations.

The original version of AltaRica and AltaRica Data-Flow are object-oriented languages. AltaRica 3.0 is a prototype oriented language. Most of the AltaRica models consider systems at plant level. As a consequence, model components have in general a unique instance. Moreover, components can be part of several hierarchies (of components), as exemplified by so-called functional chains. Prototype orientation is dramatically better suited for this type of modeling than object orientation.

AltaRica 3.0 is therefore a major release of the language. This document defines the details of AltaRica 3.0. It is not intended to learn the AltaRica language with this text. This specification can be used by programmers to implement an AltaRica tools and by model designers who want to understand the exact details of a particular language element.

AltaRica is a free language. At the moment, there is no such as thing as an “official” version of the language and even less a “standard”. The language specified in the present document is however clearly intended to serve as a solid ground for such a normalized version.

Contents

List of Figures

Chapter 1

Introduction

1.1 Scope of the Specification

AltaRica 3.0 is a language for event driven modeling of complex systems. Its semantics is defined in terms of Guarded Transition Systems (GTS). It is built on prototype-oriented constructs to facilitate both the modeling process and the reuse of modeling knowledge. Namely, an AltaRica model consists in hierarchies of components, so-called boxes, which encapsulate GTS. The semantics of the AltaRica language is specified by means of a set of rules for translating any box to a flat AltaRica structure, i.e. a GTS. This translation is called flattening in the AltaRica jargon. The key issues of AltaRica specification are:

- The definition of Guarded Transition Systems;
- The association of (stochastic) delays to events so to obtain (stochastic) timed models;
- The construction of hierarchies of boxes and the specification of flattening rules;
- The synchronization of events;
- ... and of course the syntax for the different language elements.

This specification does not present however the full AltaRica picture. Issues not addressed in the present document include:

- Best modeling practices, modeling patterns and gadgets, ...;
- Tool specific information. A variety of tools have been developed to assess AltaRica 3.0 models, including compiler to Fault Trees and to Markov Chains, generators of critical sequences, stochastic simulators, and model-checkers. Calculations performed by these tools are typically described in command files that have their own syntax and semantics. Moreover, some of these tools may interpret AltaRica 3.0 models in some specific way. For instance, compilers to Fault Trees do not usually take into account delays.

AltaRica 3.0 main language elements are reviewed in the next sections.

1.2 Notations and Grammar

Throughout this document, grammar rules are given for the various AltaRica 3.0 constructs. The syntax for these grammar rules is a simplified version of the syntax used by parser generators such as

ANTLR:

- Terminal symbols are either surrounded with quotes when they are given literally, e.g. "`class`", "`event`", or written in capital letters when they need to be defined, e.g. `FLOAT`, `IDENTIFIER` (denoting respectively floating point numbers and identifiers).
- Non terminal symbols are capitalized, e.g. `ClassDeclaration`. Their definition starts with the symbol "::=".
- Symbols are grouped using parentheses "(" and ")".
- Alternatives are denoted with the symbol "|".
- A symbol or a group of symbols followed by "∗", "+" or "?" denotes respectively any number of occurrences, at least one occurrence and at most one occurrence of that symbol or group of symbols.

For instance, the following rule depicted Figure 1.1 describes a non-empty list of "Object" separated with commas. It means lists of objects as either a unique object, followed by any number of times a comma followed by an object

```
ObjectList ::=
      Object ( `,' Object )*
```

Figure 1.1: Example of a rule.

- Keywords are reserved words and may not be used as identifiers.

1.3 Organization of the Document

The remainder of this document is organized as follow:

- Chapter 2 makes a quick tour of AltaRica 3.0.
- Chapter 3 presents Guarded Transition Systems (GTS). This chapter stays at a rather abstract level in order to keep the semantics of AltaRica 3.0 independent of implementation details.
- Chapter 4, chapter 5 and chapter 6 present respectively the lexical structure, expressions and instructions of the language. These chapters are necessary to have a complete specification of the language.
- Chapter 9 describes preprocessed elements: constants, records and functions. These elements are declared just as others, but they are essential to the language. They are actually replaced by their definition during a preprocessing phase.
- Chapter 10 presents the hierarchy of classes and blocks.
- Chapter 11 presents elements about stochastic models.

Appendices extend the specification with some additional material.

- Appendix A gives the complete E-BNF grammar of AltaRica 3.0.
- Appendix B gives the complete E-BNF grammar and XML Schema Definition of GTS.
- Appendix C recalls the history of revisions of the specification.

Chapter 2

A Quick Tour of AltaRica 3.0

2.1 Guarded Transition Systems

We shall give a formal definition of Guarded Transitions Systems (GTS) in chapter 3. Informally, a GTS is made of the following elements:

- A set of variables, so-called state variables, which describe the state of the system. These variables take their values into finite domains (such as Boolean or enumerations of symbolic constants) or infinite domains (such as integers, floating point numbers or symbolic constants).
- Another set of variables, so-called flow variables, which describe the transfer function realized by the system, i.e. what enters and goes out of the system. Flow variables take their values into domains, just as state variables. Flow variables functionally depend on state variables: once the value of the latters is set, the value of the formers is uniquely determined.
- A set of events that may occur in the system.
- A set of labeled transitions that describe how the system evolves. A transition is a triple $\langle e, G, P \rangle$, denoted as $e : G \rightarrow P$, where:
  - e is the event labeling the transition.
  - G is a Boolean condition over variables (state and flow ones), so-called the guard of the transition.
  - P is the action to be performed on state variables to calculate the new state of the system. A transition $e : G \rightarrow P$ is fireable when its guard is satisfied.
- Finally, an assertion which is used to calculate the value of flow variables from the value of state variables.

Figure 2.1 shows the finite state automaton of the GTS done by the AltaRica 3.0 code depicted Figure 2.2 and representing a repairable Pump. The model starts with the declaration of a domain named `RepairableState`, i.e. a set of two symbolic constants `WORKING` and `FAILED`. Then it declares a class (a type of box, see section 2.2 and chapter 10), which in this case is a GTS called `Pump`. This GTS contains: a state variable `s` that takes its value into the domain `RepairableState` and whose initial value is `WORKING`, two boolean flow variables `inFlow` and `outFlow` whose reset values are both `false`, two events: `failure` and `repair`, one transition for each event, and finally an assertion setting a constraint on the values of flow variables.

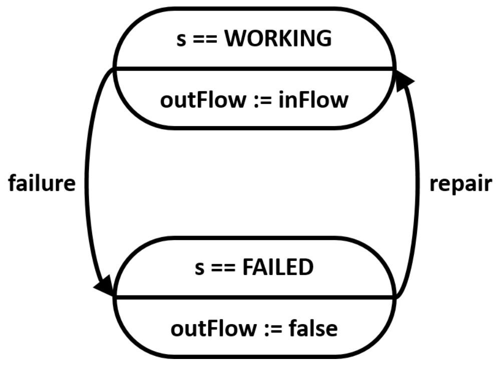


Figure 2.1: Finite state automaton for a component Pump.

```
domain RepairableState {WORKING, FAILED}

class Pump
    RepairableState s (init = WORKING);
    Boolean inFlow, outFlow (reset = false);
    event failure, repair;
    transition
        failure: s == WORKING -> s := FAILED;
        repair: s == FAILED -> s := WORKING;
    assertion
        outFlow := if s == WORKING then inFlow else false;
end
```

Figure 2.2: AltaRica 3.0 code for the component Pump.

Into GTS, state variables are initialized once for all. Their initial value is given by the attribute `init`. This attribute characterizes state variables. Flow variables are updated after each transition firing. Either their value is forced by the assertion or they take their reset value, given by the attribute `reset`. This attribute characterizes flow variables. If this class `Pump` is used in isolation, both `inFlow` and `outFlow` are systematically reset to the value `false` after each transition firing. If it is connected to other components (represented by instances of classes or blocks, see section section 2.2 and chapter 10) then the assertion propagates the value of flows (when the Pump is working). This mechanism is explained in details in chapter 3. The transition labeled with the event `failure` is fireable only when the state variable `s` is equal to `WORKING` (i.e. the guard done by the expression `s == WORKING` is true). The firing of the transition first set the value of the state variable `s` to `false`, then it updates the values of flow variables by calling the assertion.

On the AltaRica 3.0 code given Figure 2.2, keywords are colored. Although it is not required by the AltaRica 3.0 syntax, we obey in general the following notational conventions:

- Names of domains, classes, blocks and other declared elements are capitalized, e.g. `RepairableState`.
- Symbolic constants are written with upper-case letters, e.g. `WORKING`.
- State and flow variables, events and parameters are capitalized but start with a lower-case letter when their name contain more than one letter, e.g. `inFlow`, or are written with a single lower-case or upper-case letter possibly followed with digits, e.g. `s` or `V33`.

2.2 Hierarchies of Components (S2ML)

AltaRica 3.0 is a prototype-oriented language. It combines advantages of both paradigms prototype-oriented and object-oriented. A box, encapsulating a GTS with some given characteristics, can be used in two different ways. Either with the prototype-oriented paradigm by declaring this box and using it directly, namely a block. Or with the object-oriented paradigm, needing to previously define a class corresponding to the box and then to instantiate it into a block or another class.

Formally a block is a structural construct that represents a prototype, i.e. a component having a unique occurrence in the model. A class is a structural construct that defines a generic component. It is first defined and then used in the model via instantiation, i.e. cloning of a generic component. An instance of a class is also called an object. A block can be seen as a class with only one instance which is the block itself. A block is thus also called an object. So an object can represent a block or an instance of a class.

An AltaRica 3.0 model is a set of structured classes definitions and instantiations and blocks into a hierarchy. Different operations can be done between blocks and classes in order to structure an AltaRica 3.0 model: inheritance, composition, aggregation and cloning. Composition and inheritance are used to build hierarchical models organized in a tree. Aggregation and cloning are used to create hierarchical models with branches, of this hierarchy, and it is possible to share components between different branches. This operation aggregation is allowed because of the prototype-oriented paradigm. All these mechanisms are completely described into chapter 10 and the following summarizes them by using an example. Assume for instance we want to model the Cooling System pictured Figure 2.3.

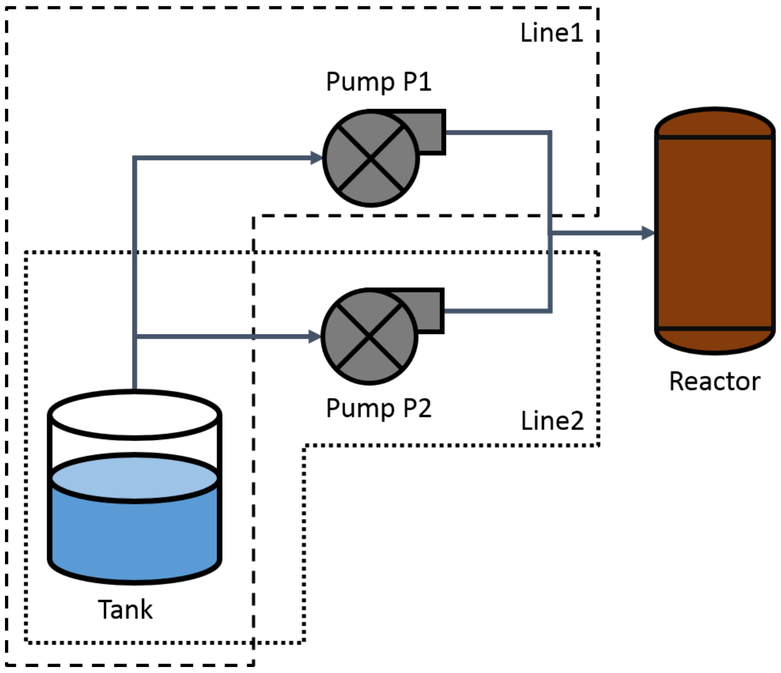


Figure 2.3: A Cooling System.

The Cooling System provides coolant from a tank to a reactor by two possible lines. These two lines are identical from a component point of view. Each one is composed of a pump and shares the tank.

2.2.1 Composition (the declaration clause)

The basic way to include the elements of a class into another class or a block is to use the composition. A block and a class can be composed of objects (other blocks or instances of classes) also with the additional constraint that this introduces no circular definitions.

When a class A or a block B is composed of an instance c of another class C, then the elements of C are added to the class A or to the block B and their names are prefixed by the name of the instance followed by a dot.

In the example given Figure 2.4, the block `Line1` contains an instance of the class `Pump` named `P1`. We say that the block `Line1` is composed of the instance `P1`. Elements of the internal instantiation `P1` can be accessed in the block `Line1` by means of the dot (‘`.`’) notation, as exemplified above with the variables `P1.s` to access to the state variable `s` of the instance `P1`. Of course, the internal class may itself be composed by classes and so on.

```
block Line1
    Pump P1;
    Boolean output (reset = false);
    // ...
    assertion
        output := P1.s == WORKING;
end
```

Figure 2.4: AltaRica 3.0 code for composition.

2.2.2 Inheritance (the extends clause)

The inheritance (found in all object-oriented programming and modeling languages) is another basic way to embed the elements of a class into another class or into a block. It is specifically dedicated for reuse or adapt an already defined class. This concept is implemented in AltaRica 3.0 via the clause extends.

A class or a block may extend another class, with the constraint that this introduces no circular definitions, e.g. a class C cannot extend a class B if the class B already extends the class C. Multiple inheritance is possible in AltaRica 3.0, although not recommended (as in all object-oriented languages).

The usage of the `extends` clause is illustrated by the AltaRica 3.0 code depicted Figure 2.5. It first defines a class `RepairableComponent` representing a generic repairable component. Then it defines the class `Pump` by extending the class `RepairableComponent` and adding the flow variables representing the input and the output streams and the corresponding assertion.

```
domain RepairableState {WORKING, FAILED}

class RepairableComponent
    RepairableState s (init = WORKING);
    event failure, repair;
    transition
        failure: s == WORKING -> s := FAILED;
        repair: s == FAILED -> s := WORKING;
end

class Pump
    extends RepairableComponent;
    Boolean inFlow, outFlow (reset = false);
    assertion
        outFlow := if s == WORKING then inFlow else false;
end
```

Figure 2.5: AltaRica 3.0 code for inheritance.

So this definition of the class `Pump` is equivalent to the one given Figure 2.2.

2.2.3 Aggregation (the embeds-as clause)

Both physical and functional aspects of a system are taken into account in Safety studies. As example for a Fault Tree the top event is in practice almost always functional, e.g. "loss of the ability to provide the coolant to the Reactor" in the example Figure 2.3; but the basic events are almost always failures of physical components. Furthermore several undesirable events are often considered for the same system. Thus a component generally contributes to several functions and each function requires several components. So the question is how to create hierarchical models with different branches of the hierarchy sharing a same component.

In AltaRica 3.0, objects (blocks and instances of classes) may belong to different branches of a hierarchical model via the clause embeds. The use of this clause is illustrated by the (partial) code given Figure 2.6. The instance `T` of the class `Tank` is firstly defined into the overall block `CoolingSystem` and then shared between the blocks `Line1` and `Line2`.

```
block CoolingSystem
    Tank T;
    block Line1
        embeds owner.T as TK;
        Pump P1;
        assertion
            P1.inFlow := TK.outFlow;
    end
    block Line2
        embeds owner.T as Tk;
        Pump P2;
        assertion
            P2.inFlow := Tk.outFlow;
    end
    // ...
end
```

Figure 2.6: AltaRica 3.0 code for aggregation.

Inside a block an embedded object is given an alias (the keyword `as` followed by an identifier). Thus it must be referenced inside of the block as this alias.

2.2.4 Cloning (the clones-as clause)

Consider again the system pictured Figure 2.3. It may be the case that creating a class for lines is not worth (for instance because this type of lines are specific to that system) or that lines aggregate some component of the system (for instance the tank). So it is impossible to declare them in a separate model. In such situation, a possible solution (to avoid duplicating the code) consists in declaring the block representing one of the two lines, then cloning this block.

In AltaRica 3.0, cloning is implemented by means of the clause clones-as. This construct is illustrated by the (partial) code given Figure 2.7. A block `Line1` is declared, then a block `Line2` that clones the block `Line1`.

```
block CoolingSystem
    Tank T;
    block Line1
        embeds T as TK;
        Pump P1;
        assertion
            P1.inFlow := TK.outFlow;
    end
    clones Line1 as Line2;
    // ...
end
```

Figure 2.7: AltaRica 3.0 code for cloning.

2.2.5 Flattening an AltaRica 3.0 model

Before any assessment, AltaRica 3.0 models are flattened i.e. reduced to a single block without sub-blocks. Let consider the class `Tank` and the complete block `CoolingSystem` into Figure 2.8. The flattened AltaRica 3.0 code for the Cooling System is given Figure 2.9. The flattened box named `CoolingSystem` presented Figure 2.9 encodes a GTS. This example makes clear that GTS are state graphs given in an implicit way (just as for instance Petri nets). The explicit state graph corresponding to the GTS of Figure 2.9 is pictured Figure 2.10. For the sake of the clarity, flow variables are not indicated on this picture. Their value is determined from the value of state variables.

```
class Tank
    Boolean isEmpty (init = false);
    Boolean outFlow (reset = false);
    event getEmpty;
    transition
        getEmpty: not isEmpty -> isEmpty := true;
    assertion
        outFlow := if not isEmpty then true else false;
end

block CoolingSystem
    Tank T;
    block Line1
        embeds T as TK;
        Pump P1;
        assertion
            P1.inFlow := TK.outFlow;
    end
    block Line2
        embeds T as Tk;
        Pump P2;
        assertion
            P2.inFlow := Tk.outFlow;
    end
    block Reactor
        Boolean inFlow (reset = false);
    end
    assertion
        Reactor.inFlow := Line1.P1.outFlow or Line2.P2.outFlow;
end
```

Figure 2.8: AltaRica 3.0 code for the Cooling System.

```
block CoolingSystem
    Boolean T.isEmpty (init = false);
    Boolean T.outFlow (reset = false);
    RepairableState Line1.P1.s, Line2.P2.s (init = WORKING);
    Boolean Line1.P1.inFlow, Line1.P1.outFlow (reset = false);
    Boolean Line2.P2.inFlow, Line2.P2.outFlow (reset = false);
    Boolean Reactor.inFlow (reset = false);
    event T.getEmpty;
    event Line1.P1.failure, Line1.P1.repair;
    event Line2.P2.failure, Line2.P2.repair;
    transition
        T.getEmpty: not T.isEmpty -> T.isEmpty := true;
        Line1.P1.failure: Line1.P1.s == WORKING -> Line1.P1.s := FAILED;
        Line1.P1.repair: Line1.P1.s == FAILED -> Line1.P1.s := WORKING;
        Line2.P2.failure: Line2.P2.s == WORKING -> Line2.P2.s := FAILED;
        Line2.P2.repair: Line2.P2.s == FAILED -> Line2.P2.s := WORKING;
    assertion
        T.outFlow := if not T.isEmpty then true else false;
        Line1.P1.inFlow := T.outFlow;
        Line2.P2.inFlow := T.outFlow;
        Line1.P1.outFlow := if Line1.P1.s == WORKING
                              then Line1.P1.inFlow else false;
        Line2.P2.outFlow := if Line2.P2.s == WORKING
                              then Line2.P2.inFlow else false;
        Reactor.inFlow := Line1.P1.outFlow or Line2.P2.outFlow;
end
```

Figure 2.9: Flattened AltaRica 3.0 code for the Cooling System.

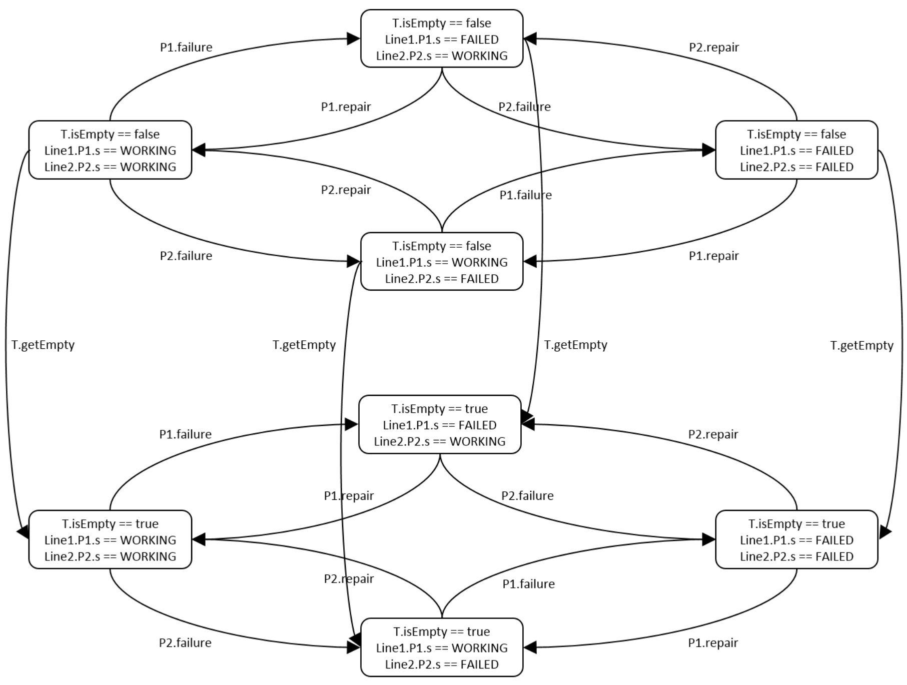


Figure 2.10: A Cooling System.

2.3 Evaluation of Actions and Assertions

Actions of transitions and assertions are instructions. Instructions in AltaRica 3.0 can take one of the following forms:

- The empty instruction "*skip*;";
- The assignment "$v := E$;", where v is a variable and E is an expression;
- The conditional instruction "$if\ C\ then\ I$", where C is a Boolean expression and I is an instruction;
- The block of instructions $I_1 ... I_n$, where I_1, ..., I_n are instructions.

Some derived forms are also accepted, such as the bidirectional assignment "v :=: w", "if-then-else" and "switch-case" instructions, but they can be rewritten using only the above instructions (see part. 6.2).

State variables can occur as the left member of an assignment only in the action of a transition. Flow variables can occur as the left member of an assignment only in the assertion. A flow variable can occur at most once as the left member of an assignment. Moreover, although some tools accept "if-then" instructions in assertions, it is recommended to use them only in actions of transitions.

Unlike the Data-Flow version, AltaRica 3.0 handles looped system and acausal component, i.e. components for which inputs and outputs are decided at run time. It introduces a specific operator to represent acausal connections: ":=:". The acausal connection of the two flow variables v and w is the instruction of the form "v :=: w" and named Bidirectional Assignment. It is equivalent to the block of the two assignments: $\{v := w; w := v;\}$. It is important to notice that this operator can only be used into assertion (to connect flow variables).

Instructions are interpreted in a different way depending they are used in the action of a transition or in an assertion. Actions of transitions are used to change locally the state of the model. Flow variables are used to model information circulating amongst the components of the model. Their value depends on the value of state variables. For this reason, it is recalculated after each transition firing.

2.3.1 Evaluation of Actions of Transitions

Let σ be the variable assignment before the firing of the transition "$e : G \rightarrow P$". Applying the instruction P to the variable assignment σ consists in calculating a new variable assignment τ as follows. We start with $\tau = \sigma$. Then,

- If P is a "*skip*;" instruction, then τ is left unchanged;
- If P is an assignment "$x := E$;" then $\tau(x)$ is set as $\sigma(E)$. If the value of x has been already modified and $\tau(x) \neq \sigma(E)$, then an error is raised;
- If P is a conditional instruction "$if\ C\ then\ I$" and if $\sigma(C)$ is true, the instruction I is applied to τ. Otherwise, τ is left unchanged.
- If P is a block of instruction $I_1 ... I_n$ then instructions I_1, ..., I_n are successively applied to τ.

It is important to note that in the above mechanism, right hand side of assignments and conditions of conditional expressions are evaluated in the context of σ. This has an important consequence: the result does not depend on the order in which instructions of a block are applied. In other words, instructions of a block are applied in parallel. For this reason, a variable cannot be assigned twice without raising an error. Also, a conditional instruction such as "$if\ x < 5\ then\ x := x + 1;$" makes sense: if $\sigma(x)$ is less than 5, then $\tau(x)$ is set to $\sigma(x)+1$, otherwise it is left unchanged. More information is done in section 3.2.1.

2.3.2 Evaluation of Assertions

Let A be the assertion and let τ be the assignment obtained after the application of the action of a transition. Applying A consists in calculating a new variable assignment (of flow variables) π as follows. We start by resetting all flow variables to their default value ($\pi(x) = reset(x)$) and with an empty set D of calculated flow variables. Then,

- If A is a "*skip*;" instruction, then π is left unchanged.
- If A is an assignment "$x := E$;" and all flow variables of E are in D, then $\pi(x)$ is set as $\pi(E)$ and x is added to D.
- If A is a conditional instruction "$if\ C\ then\ I$", all flow variables of C are in D, and if $\pi(C)$ is true, the instruction I is applied to π. Otherwise, π is left unchanged.
- If A is a block of instruction $I_1 ... I_n$ then instructions I_1, ..., I_n are repeatedly applied to π until there is no more possibility to assign a flow variable.

More information is done in section 3.2.2.

Consider the AltaRica 3.0 code given Figure 2.9 and assume:

τ(`T.isEmpty`) = `true`,

τ(`Line1.P1.s`) = `WORKING`,

τ(`Line2.P2.s`) = `WORKING`.

At first, the flow variables `T.outFlow`, `Line1.P1.inFlow`, `Line1.P1.outFlow`, `Line2.P2.inFlow`, `Line2.P2.outFlow` and `R.inFlow` are reset to their default value, i.e. `false`. Then, the following operations are performed in order.

- The instruction `T.outFlow := not T.isEmpty;` is applied, `T.outFlow` is set to `true` and added to D.
- The instruction `Line1.P1.inFlow := T.outFlow;` is applied, `Line1.P1.inFlow` is set to `true` and added to D.
- The instruction `Line1.P1.outFlow := if Line1.P1.s == WORKING then Line1.P1.inFlow else false;` is applied, `Line1.P1.outFlow` is set to `true` and added to D.
- and so on ...

2.4 Synchronizations

Just as assertions make it possible to link variables from different boxes, synchronizations make it possible to link their events (and therefore transitions). There is a difference however. Synchronizations modify events and transitions they involve.

To illustrate the synchronization mechanism, let us consider again the Cooling System pictured Figure 2.8. We can envision several reasons to synchronize events:

- The repair man can be send to repair the pumps only if both pumps are failed for only one pump is sufficient to realize the cooling function. So, events `Line1.P1.repair` and `vLine2.P2.repair` may be synchronized, i.e. compelled to occur simultaneously.
- There may a Common Cause Failure to both pumps, for instance in case of flooding. Such a failure can occur if at least one of the pumps is working. The other pump fails due to the common cause failure if it is not already failed.
- When the tank gets empty, it may cause a failure of the pumps (not already failed) for they work in unexpected conditions (no more fluid to the pump). This induced failure of the pumps may be considered, as a convenient approximation for the purpose of the study, as simultaneous with the event `T.getEmpty`. This kind of synchronization is an example of broadcasting: an emitter (the tank) emits a message (`T.getEmpty`), and the receivers (the two pumps) receive the message (`Line1.P1.failure` and `Line2.P2.failure`) if they are able to do so.

All the above type of synchronizations can be handled by means of a unique mechanism, as illustrated by the AltaRica 3.0 code presented Figure 2.11.

```
block CoolingSystem
    Tank T (getEmpty.hidden = true);
    block Line1
        embeds owner.T as TK;
        Pump P1 (repair.hidden = true);
        assertion
            P1.inFlow := TK.outFlow;
    end
    block Line2
        embeds owner.T as TK;
        Pump P2 (repair.hidden = true);
        assertion
            P2.inFlow := TK.outFlow;
    end
    block Reactor
        Boolean inFlow (reset = false);
    end
    event repair, CCF, emptying;
    transition
        repair: !Line1.P1.repair & !Line2.P2.repair
        CCF: ?Line1.P1.failure & ?Line2.P2.failure
        emptying: !T.getEmpty & ?Line1.P1.failure & ?Line2.P2.failure
    assertion
        Line1.P1.inFlow := T.outFlow;
        Line2.P2.inFlow := T.outFlow;
        Reactor.inFlow := Line1.P1.outFlow or Line2.P2.outFlow;
end
```

Figure 2.11: Synchronization mechanisms.

In AltaRica 3.0, there are two types of transitions: simple transitions as those we have seen so far, and synchronization. A synchronization is a vector of pairs (modality, event), where the modality is either "!" (meaning that the event is mandatory) or "?" (meaning that the event is optional). This vector is transformed into a simple transition at flattening time. Let "$e_1 : G_1 \rightarrow P_1$" and "$e_2 : G_2 \rightarrow P_2$" be two transitions and let "e" be a fresh event. Defining "e" as the synchronization of "e_1" and "e_2", basically produces the transition: $e : G_1 or g_2 \rightarrow \{if\ G_1\ then\ P_1; if\ G_2\ then\ P_2\}$. In other word, the synchronizing transition is fireable if at least one of the synchronized transitions is. The action of the synchronizing transition consists in firing all fireable synchronized transitions. The modality "!" forces the corresponding synchronized transition to be fireable.

The attribute "hidden" is used to discard an event (and the transitions it labels). The hidden event ceases to exist individually and the transitions they label are discarded from the model.

Figure 2.12 presents the AltaRica 3.0 flattened code of the transitions of the synchronized pumping system described Figure 2.11.

```
// ...
transition
    repair: Line1.P1.s == FAILED and Line2.P2.s == FAILED -> {
                             Line1.P1.s := WORKING;
                             Line2.P2.s := WORKING;
                                                        }
    CCF: Line1.P1.s == WORKING or Line2.P2.s == WORKING -> {
       if Line1.P1.s == WORKING then Line1.P1.s := FAILED;
       if Line2.P2.s == WORKING then Line2.P2.s := FAILED;
                                                      }
    emptying: not T.isEmpty -> {
       T.isEmpty := true;
       if Line1.P1.s == WORKING then Line1.P1.s := FAILED;
       if Line2.P2.s == WORKING then Line2.P2.s := FAILED;
                           }
    Line1.P1.failure: Line1.P1.s == WORKING -> Line1.P1.s := FAILED;
    Line2.P2.failure: Line2.P2.s == WORKING -> Line2.P2.s := FAILED;
// ...
```

Figure 2.12: Flattened AltaRica 3.0 code for the synchronized transitions of the cooling system.

The code presented Figure 2.12 can be explained as follows:

- In the synchronizing transition `repair`, both events `Line1.P1.repair` and `Line2.P2.repair` are mandatory. Therefore the guard of the synchronizing transition is the conjunction of the guard of the synchronized transitions. The action of the synchronizing transition is simplified by taking into account that both events are mandatory and therefore that their guards are both satisfied when the transition is fired.
- In the synchronizing transition `CCF`, both events are optional. The synchronization mechanism is therefore applied without simplification.
- In the synchronizing transition `emptying`, the even `T.getEmpty` is mandatory while the two other events are optional. The guard and the action of the synchronizing transition are simplified accordingly.

The finite state automaton implicitly described by the code presented Figure 2.12 is pictured Figure 2.13. Values of flow variables are omitted for the sake of the clarity.

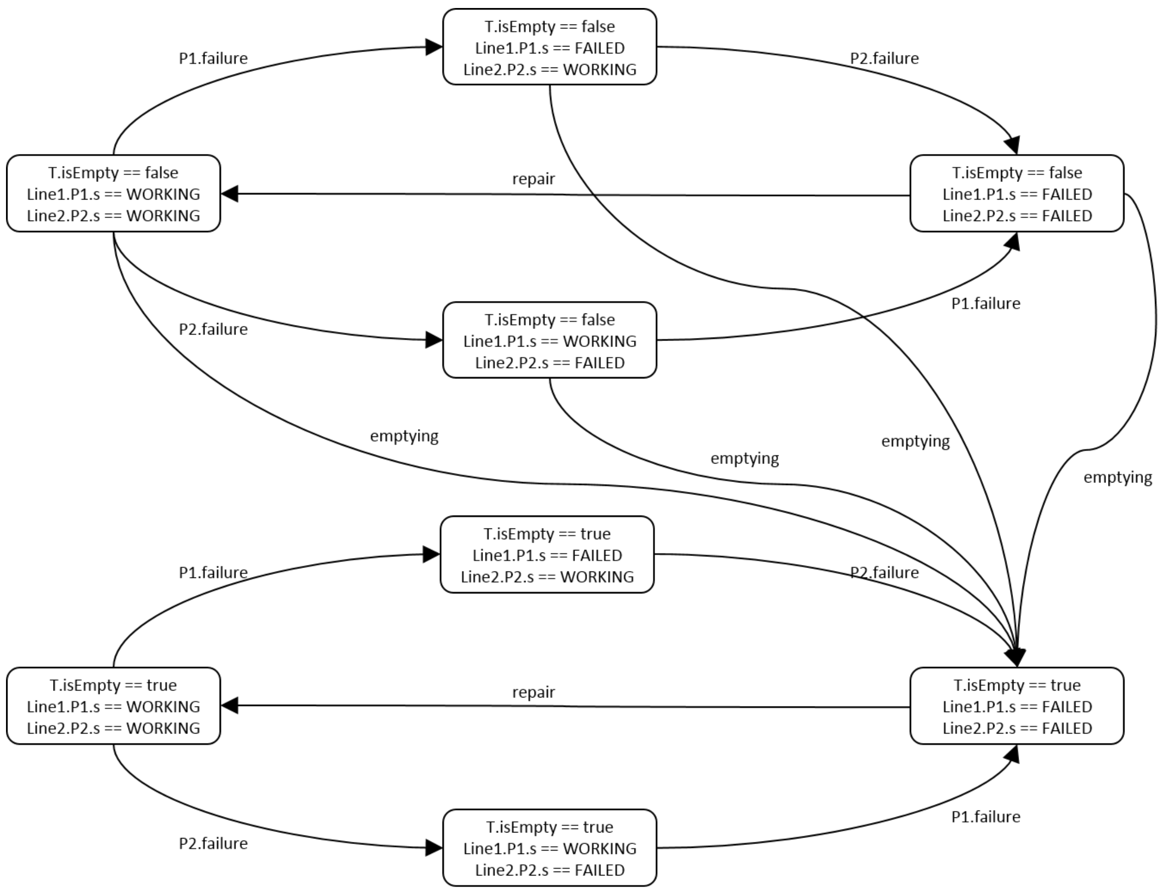


Figure 2.13: A Cooling System.

2.5 Non determinism

Nothing prevents AltaRica 3.0 transitions to be non-determinist, i.e. the same event may lead to two different states. If such phenomenon would be a bit awkward at component level, it occurs naturally with synchronizations and abstractions. As an illustration, consider again the previous example and assume we want to abstract away individual failures of pumps. The code for the cooling system would be then as pictured Figure 2.14.

```
block CoolingSystem
    Tank T;
    Pump P1, P2;
    block Reactor
        Boolean inFlow (reset = false);
    end
    event individualFailure, repair, CCF, emptying;
    transition
        repair: !P1.repair & !P2.repair
        CCF: ?P1.failure & ?P2.failure
        emptying: !T.getEmpty & ?P1.failure & ?P2.failure
        individualFailure: !P1.failure | !P2.failure
    assertion
        P1.inFlow := T.outFlow;
        P2.inFlow := T.outFlow;
        Reactor.inFlow := P1.outFlow or P2.outFlow;
end
```

Figure 2.14: Introduction of non-determinism into transitions by means of the operator "|".

The alternative operator "|" makes it possible to introduce non-determinism in transitions. It works exactly if there were several independent transitions labeled with the same event.

2.6 Parameters

Parameters are constant values which can be referred and used into the model. They are fully used to define parameters of delays (see part. 2.7 or chapter 11). Their values are constants and thus cannot be changed except at instantiation. Furthermore a parameter can only be equal to a constant value or another parameter.

The declaration and use of a parameter would be as shown Figure 2.15. The class `Pump` declares a parameter with type `RepairableState` and set to the constant value `WORKING`. This parameter is used to set the init value of the variable `s`. The block `System` instantiates two pumps: a main one named `pMain` and a spare one named `pSpare`. The parameter of `pSpare` is set to the constant value `STANDBY` when it is instantiated.

```
domain RepairableState {WORKING, FAILED, STANDBY}

class Pump
    parameter RepairableState p = WORKING;
    RepairableState s (init = p);
    // ...
end

block System
    Pump pMain;
    Pump pSpare(p = STANDBY);
end
```

Figure 2.15: Declarations and uses of Parameters.

2.7 Stochastic Models

A probabilistic structure can be put on top of GTS so to get stochastic timed models. The idea is to associate with each event the following information:

- A delay that can be deterministic or stochastic and may depend on the state. When a transition labeled with the event gets fireable at time t, a delay d is determined and the transition is actually fired at time $t + d$ if it stays fireable from t to $t + d$.
- An attribute to tell the memory policy for the event. Assume that a delay d is determined at time t_1 for a transition. Assume moreover that the transition stops to be fireable before the delay ends (say after a time $c < d$). The question is what to do when the transition gets fireable again at time t_2: either to determine a new delay d_2 (in case the policy is "restart") or to consider that the delay is now $d - c$ (in case the policy is "memory").
- A weight ("expectation") that is used to determine the probability that the transition is fired in case of several transitions should be fired at the same time.

As an illustration, consider that in our example, there is a main pump and a spare one. The spare pump is normally in standby. It is attempted to be turned on when it gets demanded, i.e. when the main pump fails. It is turned off as soon as the main pump is working again. The model for such a pump is pictured Figure 2.16.

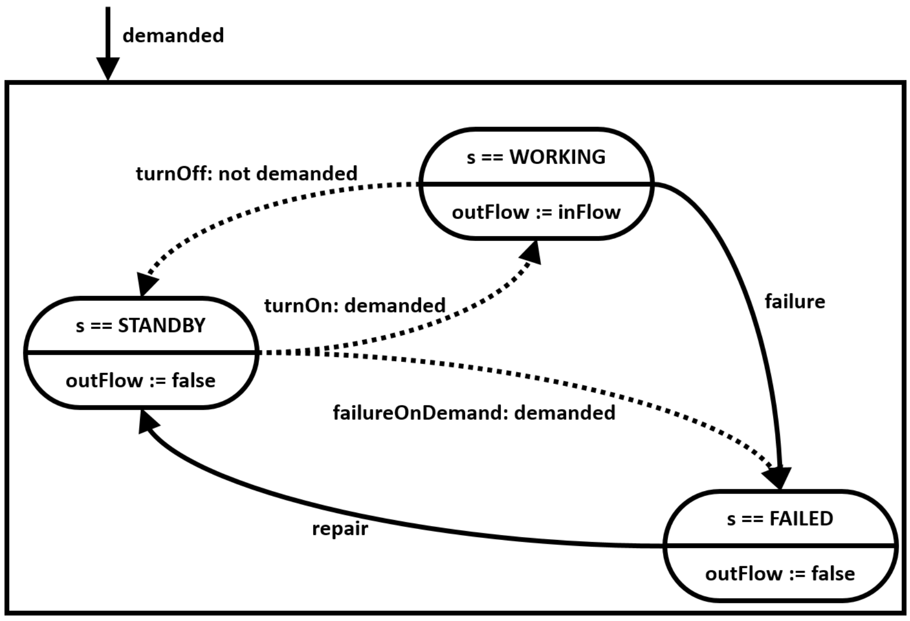


Figure 2.16: A Finite State Automaton for a spare pump.

On the GTS pictured Figure 2.16, transitions `turnOff`, `turnOn` and `failureOnDemand` are deterministic and instantaneous: they must be associated with a null delay. Transitions `failure` and `repair` are stochastic and timed. They obey typically exponential distributions with a rate λ for the failure and a rate μ for the repair. The transition `failure` can be interrupted. It is reasonable to consider it as pre-emptible, while the transition `repair` is not (it is anyway no interruptible). Finally, when the pump is attempted to be turn on, there may be a probability γ that it fails on demand, and therefore a probability $1 - \gamma$ that it starts correctly.

Figure 2.17 presents the AltaRica 3.0 code for this Guarded Transition System. This code introduces several new constructs:

- Delays are associated to events by means of the attribute "delay".
- The expression defining "delay" can be "exponential" meaning that the delay obeys the Markovian hypothesis with the given rate. By default the delay associated with an event is constant(1.0) (see chapter 11).
- Failure and repair rates as well as the probability of failure on demand are declared as parameters. Parameters are like constants except that their values can be changed when an instance of the class is created (this mechanism is described below).
- The flag indicating the memory policy for the event is simply the Boolean attribute "policy". By default this attribute is set to "restart".
- The weight of an event is defined by means of the attribute "expectation". In the above example, transitions labeled with `turnOn` and `failureOnDemand` will be systematically fireable at the same time. If no other transition is fireable, the transition labeled with `failureOnDemand` is fired with a probability $\gamma/(\gamma+(1-\gamma)) = \gamma$ and the transition labeled with `turnOn` is fired with a probability $(1-\gamma)/(\gamma+(1-\gamma)) = 1-\gamma$. By default, the expectation of a transition is 1.

```
domain SparePumpState {STANDBY, WORKING, FAILED}

class SparePump
    SparePumpState s (init = STANDBY);
    Boolean demanded (reset = false);
    Boolean inFlow (reset = false);
    Boolean outFlow (reset = false);
    parameter Real gamma = 0.2;
    parameter Real lambda = 0.001;
    parameter Real mu = 0.1;
    event stop (delay = Dirac(0.0));
    event failureOnDemand (delay = Dirac(0.0), expectation = gamma);
    event start (delay = Dirac(0.0), expectation = 1 - gamma);
    event failure (delay = exponential(lambda), policy = memory);
    event repair (delay = exponential(mu));
    transition
        stop: s == WORKING and not demanded -> s := STANDBY;
        failureOnDemand: s == STANDBY and demanded -> s := FAILED;
        start: s == STANDBY and demanded -> s := WORKING;
        failure: s == WORKING -> s := FAILED;
        repair: s == FAILED -> s := STANDBY;
    assertion
        outFlow := if s == WORKING then inFlow else false;
end
```

Figure 2.17: AltaRica 3.0 Code for a Spare Pump.

The value of attributes and parameters can be changed when the class is instanced, using the dot ("`.`") notation. For instance, if we want to embed an instance of `SparePump` that is `WORKING` in its initial state and whose failure rate is $1.23e^{-4}$, the declaration is as depicted Figure 2.18.

```
    // ...
    SparePump P (s.init = WORKING, lambda = 1.23e-4);
    // ...
```

Figure 2.18: Changing values of attributes and parameters at class instantiation.

2.8 Functions, Operators and Records

The three last constructs of AltaRica 3.0 are of less importance than the ones introduced so far, but they are convenient in many occasions.

2.8.1 Functions

AltaRica 3.0 makes it possible to declare functions. Function calls are instructions just as assignments or conditional instructions. Functions in AltaRica 3.0 are rather macro-instructions than actual functions. Recursion is not allowed. The use of functions is illustrated Figure 2.19. When the model is flattened, function calls are just replaced by the corresponding instructions.

```
domain ThreeValue {UNKNOWN, YES, NO}

function ThreeValuedOr(ThreeValue u, ThreeValue v, ThreeValue r)
    r := if u == YES or v == YES then YES
            else if u == NO and v == NO then NO
            else UNKNOWN;
end

class MyClass
    // ...
    ThreeValue in1 (reset = UNKNOWN);
    ThreeValue in2 (reset = UNKNOWN);
    ThreeValue out (reset = UNKNOWN);
    // ...
    assertion
        // ...
        ThreeValuedOr(in1, in2, out);
    // ...
end
```

Figure 2.19: Function declaration and call.

2.8.2 Operators

It is also possible to declare operators in AltaRica 3.0. They are same as functions but cannot update value of variables (state or flow ones). An operator return a typed value according to its parameters done as arguments and constants. Like for functions, operators in AltaRica 3.0 are rather macro-instructions than functions in programming languages. Recursion is also not allowed. The use of operators is illustrated Figure 2.20. When the model is flattened, operator calls are just replaced by the corresponding instructions.

```
operator Integer Increment(Integer u)
    u + 1
end

class MyClass
    // ...
    Integer var1 (reset = 0);
    Integer var2 (reset = 0);
    // ...
    assertion
        // ...
        var2 := Increment(var1);
    // ...
end
```

Figure 2.20: Operator declaration and call.

2.8.3 Records

It is often the case that several different flows are circulating between two components. It is therefore convenient to group them into a record. A record can be seen as a class with only variable declarations (no assertion, no event and therefore no transitions). Instances of records cannot be used directly into assignments but have to be used by functions (or also by operators) as illustrated on the code presented Figure 2.21.

```
domain Level {NULL, LOW, MEDIUM, HIGH}

record PipeContent
    Level water (reset = NULL);
    Level gas (reset = NULL);
    Boolean demand (reset = false);
end

function ConnectPipe(PipeContent pipe1, PipeContent pipe2)
    pipe1.water := pipe2.water;
    pipe1.gas := pipe2.gas;
    pipe1.demand := pipe2.demand;
end

block SpecialPump
    PipeContent inFlow, outFlow;
    PumpState s;
    // ...
    assertion
        // ...
        if s == WORKING then ConnectPipe(outFlow, inFlow);
    // ...
end
```

Figure 2.21: Use of records.

The assertion presented Figure 2.21 is flattened as presented Figure 2.22. Note also that in the example, if the pump is not in `WORKING` state, variables `outFlow.water`, `outFlow.gas` and `inFlow.demand` are left unassigned, i.e. are reset to their default value.

```
    // ...
    if s == WORKING then {
            outFlow.water := inFlow.water;
            outFlow.gas := inFlow.gas;
            inFlow.demand := outFlow.demand;
                         }
    // ...
```

Figure 2.22: Flattened version of the record assignment.

2.9 Observers

Observers are like flow variables, except that they cannot be used in transitions and assertions, i.e. they cannot be used to describe the behavior of a system. Rather, as their name indicates, they are quantities to be observed. They can be used or not by assessment tools. They are declared with together variables, events and parameters in very similar way parameters are. Observers are updated after each transition firing.

As an illustration, consider again the block `SpecialPump` declared Figure 2.21. We may want to put an observer on the inputs (water and gas) of the pump which takes the value:

- `HIGH` if either the water level or the gas level is `HIGH`;
- `MEDIUM` if either the water level or the gas level is `MEDIUM` but none of them is `HIGH`;
- `LOW` if either the water level or the gas level is `LOW` but none of them is `HIGH` or `MEDIUM`;
- `NULL` if both of them are `NULL`.

The declaration of such an observer would be as shown Figure 2.23. This model illustrates also the "switch" operator for expressions, which is equivalent to a cascade of "if-then-else".

```
block SpecialPump
    PipeContent inFlow, outFlow;
    PumpState s;
    observer Level inputLevel = switch {
        case inFlow.water == HIGH or inFlow.gas == HIGH : HIGH
        case inFlow.water == MEDIUM or inFlow.gas == MEDIUM : MEDIUM
        case inFlow.water == LOW or inFlow.gas == LOW : LOW
        default : NULL
        };
    // ...
end
```

Figure 2.23: Declaration of Observers.

Chapter 3

Guarded Transition Systems

The semantics of AltaRica 3.0 is defined in terms of Guarded Transition Systems (GTS). This chapter presents GTS. It stays at a quite abstract level. Relevant syntactic details will be discussed in the next chapters.

3.1 Expressions

We consider a Universe U, i.e. a denumerable set of constants C together with a finite set of operators O, a function α from O to non-negative integers that associates an arity to each operator, and a standard interpretation, denoted by $[|.|]$, that associates to each operator op a partial function from $C^{\alpha(op)}$ to C. We assume moreover that C contains the Boolean constants “true” and “false” and that O contains the Boolean operators “and”, “or” and “not” and the ternary operator “if-then-else” with their usual interpretation.

Let V a denumerable set of variables and let dom be a function that associates with each variable its domain, i.e. finite or denumerable subset of C. We build the set of expressions E as the smallest set such that:

- Constants and variables are expressions.
- If op is an operator and $E_1, ..., E_{\alpha(op)}$ are expressions then so is $op(E_1, ..., E_{\alpha(op)})$.

We denote by $var(e)$ the set of variables occurring in the expression e.

A variable assignment is a function from V to C. We say that the variable assignment σ is feasible if $\sigma(v)$ belongs to $dom(v)$ for each variable v of V. Variable assignments are lifted-up into functions from E to C in the usual way: let σ be a feasible variable assignment, let c be a constant, let op be an operator and finally let $E_1, ..., E_{\sigma(op)}$ be expressions.

- $\sigma(c) = c$
- $\sigma(op(E_1, ..., E_{\sigma(op)})) = [|op|](\sigma(E_1), ..., \sigma(E_{\sigma(op)}))$

Let σ be a possibly partial variable assignment, let v be a variable and finally let E be an expression. If σ does not give a value to a variable v we write $\sigma(v) =?$. By extension, if σ gives a value to sufficiently many variables to evaluate the expression E, we write $\sigma(E) =?$.

The calculation of $\sigma(E)$ may raise an error because operators are interpreted with only partial functions. In that case, we say that σ is not acceptable for E and we write $\sigma(E) = ERROR$. Let c be a constant, we denote by $\sigma[c/v]$ the assignment such that $\sigma[c/v](v) = c$ and $\sigma[c/v](w) = \sigma(w)$ for any other variable w.

From now, we assume that the set of variables V is the disjoint union of the set S of state variables and the set F of flow variables. Moreover, we assume that each variable has a default (or initial) value.

3.2 Instructions

Guarded Transitions Systems are built on only two instructions: conditional assignment and parallel composition. The set I of instructions is the smallest set such that:

- "*skip*;" is an instruction;
- If v is a variable and E is an expression, then the assignment "$v := E$;" is an instruction;
- If C is a (Boolean) expression and I is an instructions, then "$if\ C\ then\ I$" is an instruction;
- If I_1 and I_2 are instructions, then so is their parallel composition "$I_1; I_2$".

In the sequel, we shall consider only two types of instructions:

- Instructions in which left members of assignments are only state variables. We call these instructions actions (of transitions);
- Instructions in which left members of assignments are only flow variables. We call these instructions assertion.

The semantics of instructions depends on whether they are actions or assertions.

3.2.1 Semantics of actions (State semantics)

An action extends a given partial variable assignment τ of S by means of a total variable assignment σ of V, according to the rules given Figure 3.1 in a Structural Operation Semantics style.

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

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

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

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

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

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

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

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

Figure 3.1: Structural Operational Semantics rules that defines the semantics of actions.

Let consider an instruction P:

- (S0): if P is an empty instruction, then τ is left unchanged.
- If P is an assignment "$v := E$":
  - (S1): If τ does not give a value to v, then $\tau(v)$ is set to $\sigma(E)$.
  - (S2): If v already has a value in τ and $\tau(v) = \sigma(E)$, then τ is left unchanged.
  - (S3): If v already has a value in τ and $\tau(v) \neq \sigma(E)$, then an error is raised.
- If P is a conditional assignment "$if\ C\ then\ I$":
  - (S4): If $\sigma(C) = TRUE$, then the instruction I is applied to τ.
  - (S5): Otherwise, τ is left unchanged.
- If P is a block of instructions "$I_1, ..., I_n$":
  - (S7) - (S12): Instructions I_1,...,I_n are successively applied to τ.

This set of rules is non-deterministic. The execution of a parallel composition "$I_1; I_2$" may start with the execution of I_1 or with the execution of I_2. Although tools will probably make a systematic choice, this semantics is independent of this choice.

We denote by $Update(I, \tau)$ the function that:

1. Extends the partial variable assignment τ of S from the instruction I and the total variable assignment σ of V according to the rules presented Figure 3.1 and starting with $\tau = \emptyset$.
2. Completes τ by setting $\tau(v) = \sigma(v)$ for all variables that are not given a value by the previous step.

3.2.2 Semantics of instructions in assertion (Flow semantics)

Let A be the assertion and π the variable assignment obtained after the application of the action of a transition:

$$\pi = Update(I, \sigma)$$

Applying A to π consists in calculating a new variable assignment (of flow variables) τ as follows. We start by setting all state variables in τ to their values in π: $\forall v \in S, \tau(v) = \pi(v)$. An assertion extends a variable assignment τ which is total on S but partially partial on F, according to the rules given Figure 3.2.

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

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

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

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

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

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

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

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

Figure 3.2: Structural Operational Semantics rules that define the flow semantics of instructions.

Let D be a set of unevaluated flow variables, we start with $D = F$. Then:

- (F0): if A is an empty instruction then τ is left unchanged.

- If A is an assignment "$v := E$":
  - $\tau(E)$ can be evaluated in τ, i.e. all variables of E have a value in τ:
    (F3): If v already has a value in τ, then if $\tau(v) \neq \tau(E)$ then an error is raised.
    (F1): else τ is left unchanged.
    (F2): Otherwise, $\tau(v)$ is set to $\tau(E)$ and v is removed from D.
  - $\tau(E)$ cannot be evaluated in τ, then τ is left unchanged.
- If A is a conditional assignment "*if C then I*":
  - $\tau(C)$ can be evaluated in τ:
    (F4): If $\tau(C) = TRUE$, then the instruction I is applied to τ.
    (F5): Otherwise, τ is left unchanged.
  - Otherwise, τ is left unchanged.
- If A is a block of instructions "$\{I_1, ..., I_n\}$":
  (F7) - (FS12): Instructions I_1,...,I_n are repeatedly applied to τ until there is no more possibility to assign a flow variable.

Again, this set of rules is non-deterministic but its result does not depend on the execution order.

Let ι be the assignment that associates each variable with its initial or default value. We denote by $Propagate(I, \tau, \iota)$ the function that:

1. Extends the partial variable assignment τ by the instruction I according to the rules presented Figure 3.2.
2. Completes τ by setting each unassigned variable v to its default value $\iota(v)$.

Let ι be the total assignment that associates each variable with its initial or default value, let P be an action, let A be an assertion and let σ be a total variable assignment. We denote by $Fire(P, A, \iota, \sigma)$ the assignment defined as follows:

$$Fire(P, A, \iota, \sigma) = Propagate(A, \iota, Update(P, \sigma))$$

We denote by $\mu I(\sigma)$ the iterated application of an instruction I to a variable assignment σ. $\mu I(\sigma)$ leads to one of the following situations.

- It reaches a fixpoint.
- It loops or it diverges, i.e. runs through infinitely many different variable assignments.
- It raises an error.

There is in general no way to decide in which situation the execution of $\mu I(\sigma)$ will end up.

3.2.3 Data-Flow Assertions

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

- I is in the form "$v := E$" and $w \in var(E)$;
- I is in the form "*if C then J*" and v depends immediately on w in J;
- I is in the form "$I_1; I_2$" and v depends immediately on w either in I_1 or in I_2.

Let v and w be two variables and let I be an instruction. We say that v depends on w in the instruction I if either v depends immediately on w in I or there is a variable u such that v depends immediately on u in I and u depends on w in I.

An instruction I is said Data-Flow if no variable depends on itself in I.

Conversely to what happen for actions, the repeated application of semantics rules on an assertion may not be sufficient to evaluate all instructions. For instance, the instruction "$v := w; w := v$" cannot be evaluated. If the assertion is data-flow, then such a problem cannot occur. Not only the application of semantics rules terminates, but it exists at least one reordering of block instructions that makes it possible to evaluate all instructions in order, for instance from left to right. This order is determined thanks to the dependence relation. For an instruction which is not Data-Flow, it is required to check the consistency between evaluated variables and reset values of unevaluated ones.

3.3 Definition and Regular Semantics

A Guarded Transition Systems is a quintuple $< V, E, T, A, \iota >$, where:

- V is a set of variables. V is the disjoint union of the set S of state variables and the set F of flow variables: $V = S \cup F$.
- E is a set of events.
- T is a set of transitions, i.e. of triples $< e, G, P >$, where e is an event of E, G is a Boolean expression built on variables of V (the guard of the transition) and P is an instruction built on variables of V. For the sake of the clarity, we shall write a transition $< e, G, P >$ as $e : G \rightarrow P$.
- A is an assertion, i.e. an instruction built on variables of V.
- ι is an assignment of variables of V, so-called initial or default assignment.

A transition $e : G \rightarrow P$ is fireable in a given state σ, i.e. for a given variable assignment σ, if $\sigma(G) = true$. The firing of the transition $e : G \rightarrow P$ is performed into two steps: first, state variables are updated; second flow variables are propagated, as explained previously.

Guarded Transitions Systems are implicit representations of Kripke structures, i.e. of graphs whose nodes are labeled by variable assignments and whose edges are labeled by events. More exactly, the regular semantics of the Guarded Transitions System $< V = S \cup F, E, T, A, \iota >$ is the smallest Kripke structure $< \Sigma, \Theta >$, where Σ is a set of states and Θ is a set of transitions, such that:

- $\sigma_0 = Propagate(A, \iota, \iota) \in \Sigma$. σ_0 is the initial state of the Kripke structure.
- If $\sigma \in \Sigma$ and T contains a transition $e : G \rightarrow P$ which is fireable in σ, then $\tau = Fire(P, A, \iota, \sigma) \in \Sigma$ and $\sigma \xrightarrow{e} \tau \in \Theta$.

The calculation of $< \Sigma, \Theta >$ may raise errors. A well designed Guarded Transitions System avoids this problem.

3.4 Timed Guarded Transition Systems

A Timed Guarded Transition System is a tuple $< V = S\cup, E, T, A, \iota, delay, policy >$ where:

- $V = S\cup, E, T, A, \iota >$ is a GTS;
- $delay$ is a function from events to positive real numbers.

- *policy* is a function from events to $\{restart, memory\}$.

For the sake of the simplicity, we made "*delay*" to depend only on the event. It is however possible to have "*delay*" depending on the current state, the elapsed time since the beginning of the mission and to be stochastic.

A schedule X is a function that associates with each transition T the three following positive real numbers.

- $X.lastDate(T)$, the last date when the transition T became fireable.
- $X.firingDate(T)$, the date when the transition T is scheduled to be fired.
- $X.remainingTime(T)$, the time when the transition T stayed fireable without being eventually fired. $X.remainingTime(T)$ equals always 0 when $policy(T) = restart$.

The semantics of Timed GTS could be defined as an extended Kripke structure by internalizing dates of occurrences of events, but it is probably better to define it through the notion of run.

A run is a finite sequence of dated states and transitions:

$$(\sigma_0, d_0, X_0) \xrightarrow{T_0} (\sigma_1, d_1, X_1) \xrightarrow{T_1} ... \xrightarrow{T_{n-1}} (\sigma_n, d_n, X_n)$$

where:

- The $\sigma_i' s$ are states of the Kripke structure as defined in the previous section.
- The $d_i' s$ are dates, i.e. positive real numbers. d_i is the date at which the run entered into the state σ_i.
- The $X_i' s$ are schedules of the transitions of the GTS.
- The $T_i' s$ are transitions of the GTS.

To define the set of possible runs of a GTS we proceed in two steps: first, we define possible empty runs, i.e. runs with a single state and no transition. Then we show how to extend a possible run.

A (possible empty) run is made of a single state $(\sigma_0, 0, X_0)$, where:

- $\sigma_0 = Propagate(A, \iota, \iota)$ is defined as in the previous section;
  - X_0 is a schedule of the transitions of the GTS such that for each transition $e: G \to P$ the following properties hold:
  - $X_0.lastDate(e: G \to P) = 0$ if $e: G \to P$ is fireable in σ_0 and $+\infty$ otherwise.
  - $X_0.firingDate(e: G \to P) = delay(e)$ if $e: G \to P$ is fireable in σ_0 and $+\infty$ otherwise.
  - $X_0.remainingTime(e: G \to P) = 0$.

σ_0 is both the first and the last state of the initial run.

Now let Λ be a possible run defined as follows:

$$\Lambda = (\sigma_0, d_0, X_0) \xrightarrow{T_0} (\sigma_1, d_1, X_1) \xrightarrow{T_1} ... \xrightarrow{T_{n-1}} (\sigma_n, d_n, X_n)$$

with $n \geq 1$. Let $e: G \to P$ be a transition such that $X_n.firingDate(e: G \to P)$ is minimum, i.e. is the earliest date at which the transition is scheduled to be fired. If such a transition exists, we can extend Λ into the run K defined as follows:

$$K = (\sigma_0, d_0, X_0) \xrightarrow{T_0} (\sigma_1, d_1, X_1) \xrightarrow{T_1} ... \xrightarrow{T_{n-1}} (\sigma_n, d_n, X_n) \xrightarrow{T_n} (\sigma_{n+1}, d_{n+1}, X_{n+1})$$

where:

- $T_n = e: \ G \rightarrow P$;
- $\sigma_{n+1} = Fire(P, A, \iota, \sigma_n)$ is defined as in the previous section;
- $d_{n+1} = X_n.firingDate(e: \ G \rightarrow P)$;
- X_{n+1} is the schedule built from X_n as follows:
  For each transition $e: \ G \rightarrow P$ of the GTS we are in one of the four following cases:
  Case 1) $e: \ G \rightarrow P$ was fireable in σ_n and remains fireable in σ_{n+1}. In this case:
  - $X_{n+1}.lastDate(e: \ G \rightarrow P) = X_n.lastDate(e: \ G \rightarrow P)$;
  - $X_{n+1}.firingDate(e: \ G \rightarrow P) = X_n.firingDate(e: \ G \rightarrow P)$;
  - $X_{n+1}.remainingTime(e: \ G \rightarrow P) = X_n.remainingTime(e: \ G \rightarrow P) = 0$.

  Case 2) $e: \ G \rightarrow P$ was fireable in σ_n and is not fireable in σ_{n+1}. In this case:
  - $X_{n+1}.lastDate(e: \ G \rightarrow P) = +\infty$;
  - $X_{n+1}.firingDate(e: \ G \rightarrow P) = +\infty$;
  - $X_{n+1}.remainingTime(e: \ G \rightarrow P) = Xn.firingDate(e: \ G \rightarrow P)d_{n+1}$ if $policy(h) = memory$ and 0 otherwise.

  Case 3) $e: \ G \rightarrow P$ was not fireable in σ_n and is fireable in σ_{n+1}. In this case:
  - $X_{n+1}.lastDate(e: \ G \rightarrow P) = d_{n+1}$;
  - $X_{n+1}.firingDate(e: \ G \rightarrow P) = d_{n+1} + delay(h)$ if $policy(h) = restart$ or if $X_n.remainingTime(e: G \rightarrow P) = 0$ and $d_{n+1} + Xn.remainingTime(e: \ G \rightarrow P)$ otherwise;
  - $X_{n+1}.remainingTime(e: \ G \rightarrow P) = 0$.

  Case 4) $e: \ G \rightarrow P$ was not fireable in σ_n and is not fireable in σ_{n+1}. In this case:
  - $X_{n+1}.lastDate(e: \ G \rightarrow P) = X_n.lastDate(e: \ G \rightarrow P)) = +\infty$;
  - $X_{n+1}.firingDate(e: \ G \rightarrow P) = X_n.firingDate(e: \ G \rightarrow P)) = +\infty$;
  - $X_{n+1}.remainingTime(e: \ G \rightarrow P) = X_n.remainingTime(e: \ G \rightarrow P) = 0$.

The above development makes clear why the definition of a timed semantics for GTS in terms of Kripke structure is not appropriate: the number of possible runs is not countable for the potential outcomes of the function delay are not (except if strong restrictions are put on this function).

It is worth to note that any possible run corresponds to a path in the Kripke structure that defines the regular semantics of the GTS. The converse is indeed not necessarily true.

3.5 Stochastic Guarded Transition Systems

The timed interpretation of GTS presented in the previous section does not specify how delays are calculated. Therefore, it encompasses the case where delays are “stochastic”. It remains however to define what stochastic means. To do so, we shall introduce the notion of oracle. The idea is to delegate all the “randomness” of a run to an oracle. In this way, the GTS stays purely deterministic but its behavior depends on the outcomes of the oracle.

More formally, an oracle is an infinite sequence of real numbers comprised between 0 and 1 (included). The only operation available on an oracle is to consume its first element. This operation returns the first element and the remaining of the sequence (which is itself an oracle).

Now a Stochastic Guarded Transition System is tuple $< V = S\cup, E, T, A, \iota, \mathit{delay}, \mathit{expectation}, \mathit{policy} >$ where:

- $< V = S\cup, E, T, A, \iota >$ is a GTS;
- *delay* is a function from events and oracles to positive real numbers;
- *expectation* is a function from events to positive real numbers;
- *policy* is a function from events to $\{\mathit{restart}, \mathit{memory}\}$.

For the sake of the simplicity, we made “delay” to depend only on the event and the oracle. As previously, it is however possible that “delay” depends on the current state and the elapsed time since the beginning of the mission.

The semantics of stochastic GTS is essentially similar to the semantics of Timed GTS, with two exceptions:

- The function “delay” is called with the event and the oracle as parameters;
- When several transitions are scheduled to be fired at the same date, one is picked at random by using the oracle and according to their expectations. Namely, the probability $p(e:\ G \rightarrow P)$ to fire the transition $e:\ G \rightarrow P$ (scheduled at time t) is defined as follows:

$$p(e:\ G \rightarrow P) = \frac{\mathit{expectation}(e)}{\Sigma_{f:H \rightarrow Q\ \mathit{scheduled\ at\ time\ t}} \mathit{expectation}(f)}$$

The above mechanism completes the semantics of GTS.

Chapter 4

Lexical Structure

This chapter describes several of the basic building elements of AltaRica 3.0 such as characters and lexical units including identifiers, literals and comments. The information presented here is derived from the more formal specification presented Appendix A.

4.1 Character Set

The character set of the AltaRica 3.0 language is ASCII. Unicode characters are also authorized, but most of the AltaRica tools do not accept this extension.

As in most of the programming languages, any (positive) number of white spaces can be inserted between two lexical elements. White spaces are as follows:

```
SPACE ::=
      " " | "\t" | "\n" | "\r"
```

4.2 Comments

There are two kinds of comments in AltaRica 3.0 which are not lexical units in the models and therefore are ignored by an AltaRica 3.0 parser. The comment syntax is similar to that of C++. The following comment variants are available:

- `//` comment: Characters from `//` to the end of the line are ignored.
- `/* ... */` comments: Characters between `/*` and `*/` are ignored, including line terminators.

4.3 Keywords

Keywords are reserved words in the AltaRica 3.0 grammar and therefore are not available as identifiers for user defined constructs. Here follows a list by categories:

- Constructs: `parameter`, `domain`, `block`, `class`, `record`, `function`, `operator`, `end`, `transition`, `extends`, `embeds`, `clones`, `as`, `assertion`.
- Types: `Boolean`, `Real`, `Integer`, `Symbol`, `event`, `state`.
- Operators: `and`, `or`, `not`, `if`, `then`, `else`, `switch`, `case`, `default`, `skip`.

- Attributes: `init`, `reset`, `orientation`, `delay`, `expectation`, `policy`, `hidden`.
- Built-in: `min`, `max`, `abs`, `ceil`, `div`, `exp`, `floor`, `log`, `log10`, `mod`, `pow`, `sqrt`, `time`.
- Delays: `Dirac`, `exponential`, `Weibull`, `constant`, `uniform`.
- Literals: `true`, `false`, `restart`, `memory`.
- Others: `main`, `owner`, `parameter`, `observer`, `include`.

4.4 Identifiers and Paths

AltaRica 3.0 identifiers are sequences of letters, digits or the underscore character, which are used for naming various items in the language. Case is significant, i.e., the names "Valve" and "valve" are different.

The following BNF-like rules define AltaRica 3.0 identifiers. A full BNF definition of the AltaRica 3.0 syntax and lexical units is available Appendix A.

```
ALPHA ::=
      [a-zA-Z]

DIGIT ::=
      [0-9]

Identifier ::=
      ( ALPHA | "_" ) ( DIGIT | ALPHA | "_" )*

Path ::=
      Identifier ( "." Path )?
    | "owner" "." Path
    | "main" "." Path
    | Identifier "::" Path
```

Paths are sequences of identifiers separated with dots ("`.`") or two colons ("`::`"). Paths are used to refer elements into a model or libraries. The separator two colons ("`::`") is used to refer elements into a library. For instance `lib::Valve` means the element (a class, a record, a function, an operator or a constant) named `Valve` into the library named `lib`. Both separators can be mixed, as `lib::Valve.leftFlow`.

AltaRica 3.0 provides actually powerful mechanisms to describe paths into a model. These mechanisms are extensively used for aggregation (see part 10.2.3) and cloning (see part 10.2.4).

First, it is possible to refer to objects from anywhere to anywhere in the block hierarchy by means of relative paths and the "owner" mechanism. The keyword `owner` is used in paths to refer to the parent block of the current block it is called. For instance in Figure 4.1 within the block `S.B.B2`, the cloned block `S.A.A1` is referred with a relative path as `owner.owner.A.A1`.

Second, it is possible to refer to objects by means of absolute paths by means of the "main" mechanism. An absolute path starts with the keyword `main` which designates the outmost block, i.e. the model itself (whatever the name of this block). As sketched in Figure 4.1 within the block `S.B`, the embedded block `S.A` is referred with an absolute path as `main.A`.

```
block S
    block A
        block A1
            // ...
        end
    end
    block B
        embeds main.A as A
        block B2
            clones owner.owner.A.A1 as A1;
        end
    end
end
```

Figure 4.1: Relative and absolute paths using the “owner” and the “main” constructs.

Previous examples show how to use relative an absolute paths to refer to blocks. They are design to refer to objects, so also to instances of classes. Finally it would be thus possible, although also quite awkward, to refer to others elements into objects, e.g. variables or parameters, as sketched in Figure 4.2.

```
block S
    block A
        Boolean s (init = false);
    end
    block B
        // ...
        block B1
            event e;
            transition
                e: main.A.s -> owner.owner.A.s := false;
        end
        // ...
    end
end
```

Figure 4.2: Relative and absolute paths for elements in objects.

4.5 Literals

Literals are the atomic values that have different forms depending on their type.

4.5.1 Booleans

The two Boolean literal values are true and false. They can be written only in lower case.

```
Boolean ::=
     "true" | "false"
```

4.5.2 Integers

An integer is a sequence of digits. The minimal recommended number range is from −2147483648 to +2147483648.

```
Integer ::=
      DIGIT+
```

The previous rules only considers positive integer numbers. For negative ones, we use the rules to construct expression with the minus operator '−'; also done for positive numbers with the plus operator '+'.

4.5.3 Reals

A real (i.e. a floating point number) is expressed as a decimal number in the form of a sequence of digits optionally followed by a point, optionally followed by a sequence of digits. At least one digit must be present. The decimal number is optionally followed by an exponent. The exponent is indicated by a character 'E' or 'e', followed by an optional sign ('+' or '−') and one or more digits. The recommended range is that of IEEE double precision floating point numbers, for which the largest representable positive number is $1.7976931348623157E+308$ and the smallest positive number is $2.2250738585072014E-308$.

```
Real ::=
      DIGIT+ ( "." DIGIT+ )? Mantissa?

Mantissa ::=
      ( "e" | "E" ) ( "+" | "-" )? DIGIT+
```

Like rules for integer numbers, the previous rules only considers positive real numbers. We use the rules to construct expression with the minus operator '−' to consider negative real numbers ; also done for positive numbers with the plus operator '+'.

4.5.4 Literal values

Finally the set of literal values is composed of the previous atomic forms and the identifiers.

```
LiteralValue ::=
      Boolean | Integer | Real
    | LiteralAttributeValue
    | SymbolicConstant

LiteralAttributeValue ::=
      "restart" | "memory"

SymbolicConstant ::=
      Identifier
```

The keywords `restart` and `memory` are used with the attribute `policy` of events (see part 3.4 or chapter 11).

Chapter 5

Expressions

AltaRica 3.0 models contain symbolic, Boolean and numerical expressions. This chapter describes syntax, types and evaluation rules for expressions, built-in mathematical operators and functions.

5.1 Types

Types in AltaRica 3.0 are divided onto two kinds: base types and reference types. Base types are “Boolean” (Boolean constants), “Real” (floating point numbers), “Integer” (integers, considered as a subset of “Real”), “Symbol” (all symbol or identifier constants). A reference type is either a reference to a “domain” (see chapter 8): i.e. an enumeration which is a finite set of symbolic constants; or references to a record or a class (used for instantiation).

The grammar rules that define AltaRica 3.0 types are as follow:

```
Type ::=
      BaseType
    | TypeReference

BaseType ::=
      "Boolean" | "Integer" | "Real" | "Symbol"

TypeReference ::=
      DomainReference
    | RecordReference
    | ClassReference

DomainReference ::=
      Path

RecordReference ::=
      Path

ClassReference ::=
      Path
```

5.2 Relational and Logical Operators

AltaRica 3.0 supports the standard set of relational and logical operators, all of which produce the standard Boolean values true or false.

Relation Symbol	Semantics
`==`	equality within expressions
`!=`	inequality
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to

A single equals sign `=` is never used in relational expressions, only in declarations.

The following logical operators are defined:

Logical Operator	Semantics
`not`	negation (unary operator)
`and`	logical and
`or`	logical or

5.3 Arithmetic Expressions

AltaRica 3.0 supports the standard arithmetic operators, all of which produce either integers, if all of their arguments are integers, or reals when at least one of their arguments is real. The division is an exception to this rule: it produces always a real number. To get integer division, one should use the cast built-in operator “Integer” to the result.

The arithmetic operators are as follows:

Arithmetic Operator	Semantics
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	(real) division
`+`	plus (unary operator)
`-`	minus (unary operator)

5.4 Built-in Operators

Built-in operators of AltaRica 3.0 have the same syntax as a function call. The following table summarizes the available built-in operators together with the type(s) of their argument(s) and their semantics:

Built-in Arithmetic Operator	Semantics
`min(real,...,real)`	smallest value between a finite set of values
`max(real,...,real)`	biggest value between a finite set of values
`abs(real)`	absolute value
`ceil(real)`	smallest integer bigger than or equal to the argument
`div(integer,integer)`	algebraic quotient with any fractional part discarded
`exp(real)`	exponential base e
`floor(real)`	biggest integer smaller than or equal to the argument
`log(real>0)`	natural (base e) logarithm
`log10(real>0)`	base 10 logarithm
`mod(integer,integer)`	modulus
`pow(real,real)`	power
`sqrt(real)`	square root
`#(bool,...,bool)`	count the number of true values
`Real(integer)`	cast any boolean or numerical value to a real
`Integer(real)`	cast any boolean or numerical value to an integer

5.5 Conditional Expressions

5.5.1 If-then-else

The “if-then-else” expression has its intuitive semantics. Note that the ’else’ branch is mandatory.

5.5.2 Switch

The “switch” expression is equivalent to a cascade of “if-then-else”. Its syntax is illustrated by the following example:

```
output := switch {
    case u == YES and v == YES: YES
    case u == NO and v == NO: NO
    default: UNKNOWN
                 };
```

The above expression is equivalent to the following “if-then-else” cascade:

```
output := if u == YES and v == YES then YES
          else if u == NO and v == NO then NO
          else UNKNOWN;
```

Again, note that the “else” branch is mandatory.

5.6 Evaluation Order, Operator Precedence and Associativity

AltaRica 3.0 tools are free to simplify and rewrite expressions and not to evaluate them if these actions (or absence of actions) do not influence the result (e.g. short-circuit evaluation of Boolean expressions).

Operator precedence determines the order of evaluation of operators in an expression. An operator with higher precedence is evaluated before an operator with lower precedence in the same expression. The following table presents operators in order of precedence from highest to lowest, as derived from the AltaRica Data-Flow grammar:

Group	Syntax	Arity	Examples
(postfix) access operator	`.`	2	`A.s`
(postfix) function call	`()`	depends on the definition	`div(8,A.s)`
cast to real	`Real`	1	`Real(A.s)`
cast to integer	`Integer`	1	`Integer(A.s)`
Multiplication or division	`* /`	n-ary (left associative)	`3*A.s`
Addition or subtraction	`+ -`	n-ary (left associative)	`3*A.s + 4`
plus or minus	`+ -`	1	`-A.s`
logical relation	`== != <= >= < >`	2	`A.s == failed`
negation	`not`	1	`not A.s`
conjunction	`and`	n-ary (left associative)	`A.s and input`
disjunction	`or`	n-ary (left associative)	`A.s or input`
conditional	`if-then-else`	3	`if A.s then in else 0`

5.7 Wrap-Up

The grammar rules that define the syntax of expressions is the following. The rules `DelayName` and `Curve`, defined into the previous sub-grammar will be used and explain later (see chapter 11).

```
Expression ::=
      "(" Expression ")"
    | Expression NOperator Expression
    | UOperator Expression
    | OperatorCall
    | IfThenElseExpression
    | SwitchExpression
    | Curve
    | LiteralExpression
```

```
NOperator ::=
      "or" | "and"
    | "==" | "!=" | "<" | "<=" | ">" | ">="
    | "+" | "-" | "*" | "/"

UOperator ::=
      "not" | "-" | "+"

OperatorCall ::=
      OperatorName "(" ( Expression ( "," Expression )* )? ")"

OperatorName ::=
      Path
    | BuiltinName
    | DelayName

BuiltinName ::=
      "min" | "max" | "abs" | "ceil" | "div" | "exp" | "floor"
    | "log" | "log10" | "mod" | "pow" | "sqrt" | "#"
    | "Real" | "Integer"

DelayName ::=
      "Dirac" | "exponential" | "Weibull" | "constant" | "uniform"
```

```
IfThenElseExpression ::=
      "if" Expression "then" Expression "else" Expression

SwitchExpression ::=
      "switch" "{"
        ( "case" Expression ":" Expression )*
        "default" ":" Expression
      "}"

Curve ::=
      "[" Real ":" Real ( "," Real ":" Real )* "]"
```

```
LiteralExpression ::=
      VariableReference
    | ParameterReference
    | ObserverReference
    | LiteralValue

VariableReference ::=
      Path

ParameterReference ::=
      Path

ObserverReference ::=
      Path
```

Chapter 6

Instructions

Instructions play a central role in AltaRica 3.0 models: they are used both to described actions (post-condition) of transitions and assertions. This chapter describes mainly their syntax.

6.1 Syntax

There are fundamentally four instructions: skip, assignments, if-then instructions and blocks.

- skip is an instruction that does nothing;
- Assignments are instructions to give a variable the value of an expression;
- If-then instructions are instructions whose execution is conditioned by the value of an expression;
- Blocks are sets of instructions that are conceptually executed in parallel.

Other instructions are derived from those three.

The grammar rules that define the syntax of instructions is the following:

```
Instruction ::=
      "skip" ";"
    | VariableReference ":=" Expression ";"
    | VariableReference ":=:" VariableReference ";"
    | "if" Expression "then" Instruction ( "else" Instruction )?
    | "switch" "{"
        ( "case" Expression ":" Instruction )*
        "default" ":" Instruction
      "}"
    | "{" ( Instruction )+ "}"
    | FunctionCall

FunctionCall ::=
      FunctionName "(" ( Expression ( "," Expression )* )? ")"

FunctionName ::=
      Path
```

In the context of an action of a transition, left members of assignments must be state variables only. In the context of an assertion, left members of assignments must be flow variables only.

6.2 Operational Semantics

The operational semantics of instructions depends on whether they are used in an action (state semantics) or an assertion (flow semantics). These semantics are defined chapter 3 for the four fundamental instructions: “skip”, “assignment”, “if-then” and “composition”. The other instructions can be rewritten using only these four fundamental instructions:

- The following bidirectional assignment

```
// ...
v :=: w;
// ...
```

  is equivalent to the following block of two assignments

```
// ...
{
    v := w;
    w := v;
}
// ...
```

- The following “if-then-else” instruction

```
// ...
if C then I else J
// ...
```

  is equivalent to the following block of two conditional instructions

```
// ...
{
    if C then I
    if not C then J
}
// ...
```

- The following “switch-cases” instruction:

```
// ...
switch {
         case C1 : I1
         case C2 : I2
         // ...
         case Ck : Ik
         default : J
       }
// ...
```

is equivalent to the following block of conditional instructions

```
// ...
{
    if C1 then I1
    if not C1 and C2 then I2
    // ...
    if not C1 and not C2 /* ... */ and Ck then Ik
    if not C1 and /* ... */ and not Ck then J
}
// ...
```

- The assignment `r := s;` where `r` and `s` are records (of the same type) is expanded pair wisely, as explained Section 9.4.
- Function calls are expanded as explained Section 9.2.

Chapter 7

Attributes

A few model elements use the notion of attribute in order to define some features: an initial value of a variable or a delay of an event.

```
Attributes ::=
      "(" Attribute ( "," Attribute )* ")"

Attribute ::=
      AttributeType "=" Expression

AttributeType ::=
      Path
    | ( Path "." )? AtomicAttributeType

AtomicAttributeType ::=
      "init" | "reset"
    | "delay" | "expectation" | "policy" | "hidden"
```

The following presents different uses of attributes:

```
class Component
    // ...
    Boolean mode (init = true);
    // ...
    event changeMode;
    // ...
end

block System
    // ...
    Component p (mode.init = false, changeMode.delay = Dirac(10.0));
    // ...
end
```

The classical use is to define default values of elements. The Boolean variable `mode` of the class `Component` is initialized to the value `true`. It means that when one instantiates this class into another one or a block, the variable `mode` is set to `true` at initialization.

an other use is to overload default values of elements. Into the block `System`, an instance named `p` of the class `Component` is defined. The default value of the variable `mode` is overloaded to `false`.

Finally attributes can be added at instantiation. The instance `p` of the class `Component` into the block `System` defines an attribute `delay` to the event `changeMode` set to the value `Dirac(10.0)`. It means that only for this instance the event `changeMode` has a delay.

Some attributes associated with variables and events are mandatory, e.g. `init` or `reset` for variables. Some others are optional and default values are set to them, e.g. `delay`, also for events, set to the default value `constant(1.0)`, or `hidden` for events set to the default value `false`. Finally others can be required by a specific tool. AltaRica 3.0 is flexible enough to accept that.

In previous version of AltaRica Data-Flow, flow variables (now introduced by the attribute `reset`) had to be declared `in`, `out` or `private` (meaning local). Such an annotation is not required for it can be easily deduced from the code. However, some tools may need it. For the sake of the uniformity it is advised to use the attribute `mode` with value `IN`, `OUT` or `PRIVATE` to encode this information (see Figure 7.1 for an example of such use).

```
// ...
    Boolean upperFlow (reset = false, mode = IN);
// ...
```

Figure 7.1: Using the “mode” attribute with flow variables.

Note however that most of the tools won’t perform any check on these attributes required by a specific tool.

Chapter 8

Domains

Domains are named sets of symbolic constants. The grammar rules for domain declarations are as follows:

```
DomainDeclaration ::=
      "domain" Path "{" SymbolicConstants "}"

SymbolicConstants ::=
      SymbolicConstant ( "," SymbolicConstant )*

SymbolicConstant ::=
      Identifier
```

Declared domains can be used as a type for variables anywhere in the model. For instance:

```
domain Level {NULL, LOW, MEDIUM, HIGH}

class Tank
    // ...
    Level l (init = MEDIUM);
    // ...
end
```

Domains can share the same symbolic constants, but variables must take their values within the declared domain. For instance, the following model depicted Figure 8.1 has an error. The `NonReparaibleSource` class, extending the `NonReparaibleComponent`, tries to test the value `IN_REPAIR` to the variable `s` which type is `NonRepairableState`.

```
domain NonRepairableState {WORKING, FAILED}
domain RepairableState {WORKING, FAILED, IN_REPAIR}

class NonRepairableComponent
    // ...
    NonRepairableState s (init = WORKING);
    // ...
end

class RepairableComponent
    // ...
    RepairableState s (init = WORKING);
    // ...
end

class NonRepairableSource
    extends NonRepairableComponent;
    // ...
    Boolean outflow (reset = false);
    // ...
    assertion
        outflow := if s == IN_REPAIR or s == FAILED then false else true;
    // ...
end
```

Figure 8.1: Domains sharing symbolic constants.

Chapter 9

Preprocessed Elements

A few model elements can be declared beyond classes or blocks: constants, records, functions and operators. All these elements are resolved, i.e. replaced by regular code, by a preprocessing phase. This chapter presents the syntax and the semantics of their declarations.

9.1 (External) Parameters

It is possible to name numerical constants as an external parameter. External means outside any classes or blocks. The grammar rule for external parameter declarations is as follows:

```
ExternalParameterDeclaration ::=
      "parameter" Type Path "=" Expression ";"
```

It is the same as the one for parameters included into classes or blocks. Declared external parameters can be used in expressions anywhere in the model. For instance:

```
parameter Integer ONE = 1;

class Pipe
    // ...
    Integer flow (reset = ONE);
    // ...
    assertion
        // ...
        if in1 == ONE then // ...
        // ...
        out := ONE;
        // ...
end
```

The previous AltaRica 3.0 code is replaced by:

```
class Pipe
    // ...
    Integer flow (reset = 1);
    // ...
    assertion
        // ...
        if in1 == 1 then // ...
        // ...
        out := 1;
        // ...
end
```

9.2 Functions

Functions are closer to macro-definitions of the C language than to functions of programming languages. They can be seen as a convenient mean to write complex instructions. The grammar rules for function declarations are as follows:

```
FunctionDeclaration ::=
      "function" Path "(" Arguments? ")"
        Instruction+
      "end"

Arguments ::=
      Argument ( "," Argument )*

Argument ::=
      Type Identifier
```

The Body of the function, i.e. the instruction, should depend only on arguments of the function. Declared functions can be used as an instruction anywhere in the model. The body of the function is substituted for the function call (after the renaming of arguments). Assume for instance the function `Adder` is defined and called as follows:

```
function Adder(Boolean in1, Boolean in2, Boolean carryIn,
        Boolean out, Boolean carryOut)
    out :=     (not in1 and not in2 and carryIn)
            or (not in1 and in2 and not carryIn)
            or (in1 and not in2 and not carryIn)
            or (in1 and in2 and carryIn);
    carryOut := (in1 and in2)
            or (in1 and carryIn)
            or (in2 and carryIn);
end

class MyClass
    // ...
    Boolean left, right, up, down, low, high (reset = false);
    // ...
    assertion
        // ...
        Adder(left, right, up and down, low, high);
        // ...
end
```

The call `Adder(left, right, up and down, low, high)` will be expanded as follows:

```
// ...
{
    low :=  (not left and not right and (up and down))
        or (not left and right and not (up and down))
        or (left and not right and not (up and down))
        or (left and right and (up and down));
    high := (left and right)
        or (left and (up and down))
        or (right and (up and down));
}
// ...
```

9.3 Operators

Like for functions, operators are a convenient mean to write complex expressions. The grammar rules for operator declarations are as follows:

```
OperatorDeclaration ::=
      "operator" Type Path "(" Arguments? ")"
        Expression
      "end"

Arguments ::=
      Argument ( "," Argument )*

Argument ::=
      Type Identifier
```

The Body of the operator, i.e. the expression, should depend only on arguments of the operator. Declared operators can be used as an expression anywhere in the model. The body of the operator is substituted for the operator call (after the renaming of arguments). Assume for instance the operator `Opposite` is defined and called as follows:

```
operator Integer Opposite(Integer in)
    - in
end

class MyClass
    // ...
    Integer var1, var2 (reset = 0);
    // ...
    assertion
        // ...
        var2 := Opposite(var1);
        // ...
end
```

The sub-part of the assertion calling `Opposite(var1)` will be expanded as follows:

```
        // ...
        var2 := - var1;
        // ...
```

9.4 Records

Records look like classes or blocks except first that they cannot contain event (therefore no transition) and assertion. The interest of records stands in that they can be used in to connect several flow variables. They have to be used by means of function (and also operators) in order to define the connection (instruction). They cannot be used directly into an action or an assertion. In both cases, variable must be of the same type.

The grammar rules for record declarations are as follows:

```
RecordDeclaration ::=
      "record" Path
        RecordDeclarationClause+
      "end"

RecordDeclarationClause ::=
      ExtendsClause | VariableDeclaration

ExtendsClause ::=
      "extends" Path Attributes? ";"

VariableDeclaration ::=
      Type Path ( "," Identifier )* Attributes? ";"
```

During the preprocessing phase, records are expanded. If two records are tested for equality, the members are tested pair wisely and the tests are and-ed. For instance:

```
domain Level {NULL, LOW, MEDIUM, HIGH}

record PipeContent
    Level water (reset = NULL);
    Level gas (reset = NULL);
    Boolean demand (reset = false);
end

function ConnectPipe(PipeContent pipe1, PipeContent pipe2)
    pipe1.water := pipe2.water;
    pipe1.gas := pipe2.gas;
    pipe1.demand := pipe2.demand;
end

block SpecialPump
    PipeContent inFlow, outFlow;
    PumpState s;
    // ...
    assertion
       // ...
        if s == WORKING then ConnectPipe(outFlow, inFlow);
    // ...
end
```

The assertion presented in the above is flattened as presented after. Note also that in the example, if the pump is not in `WORKING` state, variables `outFlow.water`, `outFlow.gas` and `inFlow.demand` are left unassigned, i.e. are reset to their default value.

```
// ...
    if s == WORKING then {
            outFlow.water := inFlow.water;
            outFlow.gas := inFlow.gas;
            inFlow.demand := outFlow.demand;
                         }
// ...
```

9.5 Include Directive

It is sometimes convenient to split the model into several files. AltaRica 3.0 does not support yet the concept of module or package. It is however possible to include a file (and all declarations it contains) into another by means of the directive `include`. E.g.:

```
include "../MyModels/MyPump.alt"
```

Chapter 10

Blocks and Classes

An AltaRica model is a collection of model elements: blocks, classes, constants, domains, records and functions. The fundamental elements are classes and blocks. Model elements can be used before they are declared.

A block or a class declaration is made of three parts: a first part to declare its internal elements (variables, events, etc.); a second part in which transitions are declared and finally a third part for the assertion. The grammar of block and class declarations is given Figures 10.1.Internal elements of a block or a class can be declared in any order, but the must be declared before transitions and assertions. In the grammar given Figures 10.1 to 10.4, transitions must be declared before assertions. Some tools may accept the reverse order as well.

The declaration of internal elements of blocks and classes differs according to considering a block or a class. Some of them are common: inheritance (the rule 'ExtendsClause'), variables, events, parameters, observers and definition of sub-blocks. The others are considered only for blocks: aggregation (the rule 'EmbedsClause') and cloning (the rule 'ClonesClause').

Variables, events, parameters and observers belong to the same namespace. In other words, it is not allowed to name a variable and an event (for instance) alike. The same remark applies to records, functions and classes.

Attributes can be declared with variables and events. Some attributes have predefined meaning, e.g. `init` or `reset` for variables and `delay`, `expectation` and `policy` for events. It is also possible to define tool-specific attributes. The value of the attribute `delay` is very specific for it may involve built-in probability distributions. These built-ins are presented chapter 11.

The definition of a parameter cannot involve variables or observers. It may involve other parameters, under the condition that this introduces no circular definitions.

The definition of an observer can involve variables, parameters and other observers, under the condition that this introduces no circular definitions.

The same event may label several transitions. An event `e` declared in a class `C` can be used as label of transitions of another class `D` that embeds an instance `c` of the class `C`. This transition labeled with `c.e` is just added to the other transitions labeled with `c.e` coming from the definition of `C`.

Actions (post-conditions) of transitions can only assign state variables. Assertions can only assign flow variables.

```
BlockDeclaration ::=
      "block" Path
        ( BlockClause )*
      "end"

ClassDeclaration ::=
      "class" Path
        ( ClassClause )*
      "end"

BlockClause ::=
      BlockDeclarationClause
    | TransitionsClause
    | AssertionClause

ClassClause ::=
      ClassDeclarationClause
    | TransitionsClause
    | AssertionClause

BlockDeclarationClause ::=
      ClassDeclarationClause
    | EmbedsClause
    | ClonesClause

ClassDeclarationClause ::=
      ExtendsClause
    | VariableDeclaration
    | StateDeclaration
    | EventDeclaration
    | ParameterDeclaration
    | ObserverDeclaration
    | BlockDeclaration
```

Figure 10.1: Grammar rules for the definition of blocks and classes.

```
EmbedsClause ::=
      "embeds" Path "as" Identifier ";"

ClonesClause ::=
      "clones" Path "as" Identifier Attributes? ";"

ExtendsClause ::=
      "extends" Path Attributes? ";"
```

Figure 10.2: Grammar rules for the definition of blocks and classes.

```
VariableDeclaration ::=
      Type Path ( "," Identifier )* Attributes? ";"

StateDeclaration ::=
      "state" Type Path ( "," Identifier )* Attributes? ";"

EventDeclaration ::=
      "event" Path ( "," Identifier )* Attributes? ";"

ParameterDeclaration ::=
      "parameter" Type Path "=" Expression ";"

ObserverDeclaration ::=
      "observer" Type Path "=" Expression ";"
```

Figure 10.3: Grammar rules for the definition of blocks and classes.

```
TranstionsClause ::=
      "transition" TranstionsDeclaration+

TransitionDeclaration:
      Path ":" Transition ( "|" Transition )*

Transition ::=
      IndividualTransition ( "&" IndividualTransition )*

IndividualTransition ::=
      Expression "->" Instruction
    | SynchronizedEvent

SynchronizedEvent ::=
      "!" Path
    | "?" Path

AssertionClause ::=
      "assertion" Instruction+
```

Figure 10.4: Grammar rules for the definition of blocks and classes.

10.1 Structural constructs

AltaRica 3.0 provides constructs to build hierarchical models, i.e. to organize models as hierarchies of nested components. There are two concepts to structure models: block and class.

10.1.1 Blocks

Block is a structural construct that represents a prototype, i.e. a component having a unique occurrence in the model.

Let's consider the Cooling System pictured Figure 2.3. Figure 10.5 shows the structural part of the AltaRica 3.0 code, which represents the hierarchical structure of this system. This (partial) model contains a hierarchy of nested blocks. Each block is unique. The behavior of each component should be defined individually.

```
block CoolingSystem
    block T
        // ...
    end
    block Line1
        block P1
            // ...
        end
        // ...
    end
    block Line2
        block P2
            // ...
        end
        // ...
    end
    block Reactor /* ... */ end
    // ...
end
```

Figure 10.5: Illustration of block usage.

Blocks are similar to prototypes from prototype-based programming languages. They are used to represent components having unique occurrences in the model. Since the model of the whole system is unique, it is always represented by a block. A block can also be called an “object”.

10.1.2 Classes

If several sub-components of a system are identical (e.g. `P1` and `P2` of the Cooling System), which is generally the case, duplicate models is both tedious and error prone (copy and paste, update, etc.). The idea is to define a generic component that can be then instantiated several times in the model. The definition of such a generic component is achieved via a class.

Class is a structural construct that defines a generic component. It is used in the model via instantiation, i.e. cloning of a generic component. An instance of a class is also called an “object”.

Instead of using blocks to define the behavior of individual components, classes are used to represent them. They are then instantiated in the model as shown by the code given Figure 10.6 for the tank `T` and the pumps `P1` and `P2`.

```
class Pump
    // ...
end

class Tank
    // ...
end

block CoolingSystem
    Tank T;
    block Line1
        Pump P1;
        // ...
    end
    block Line2
        Pump P2;
        // ...
    end
    block Reactor
        // ...
    end
    // ...
end
```

Figure 10.6: Illustration of class usage.

AltaRica 3.0 classes are similar to classes in object-oriented programming languages. They are used to represent stabilized knowledge, the so-called “on-the-shelf” components. They are stored in the libraries and can be reused by instantiation.

10.2 Structural operations

Models can be organized into hierarchies of nested components by means of four operations: composition, inheritance, aggregation and cloning.

10.2.1 Composition (the declaration clause)

Composition allows the creation of hierarchies of nested components. Components are objects and thus may be of two sorts: blocks and instances of classes.

The declaration of a class into a class or a block is similar to declare primitive variables and is done by the following rule:

```
VariableDeclaration ::=
      Type Identifier ( "," Identifier )* Attributes? ";"
```

In the example given Figure 10.6, the block `Line1` contains an instance of the class `Pump` named `P1`. We say that the block `Line1` is composed of the instance `P1`. Similarly, the block `System` is composed of the instance `T` and the blocks `Line1`, `Line2` and `Reactor`.

Blocks and classes can be composed of other blocks and instances of classes (see Figure 40). These compositions are done with the additional constraint that this introduces no circular definitions, e.g. a class C cannot contain an instance of a class B if B already contains an instance of C.

10.2.2 Inheritance (the extends clause)

Besides composition, inheritance is another way to embed the elements of a class into another class or into a block.

The inheritance mechanism (found in all object-oriented programming and modeling languages) represents this type of relation between components. It is implemented in AltaRica 3.0 via the extends clause.

A class may extend another class, a block may extend a class, with an additional constraint that this introduces no circular definitions, e.g. a class C cannot extend a class B if the class B already extends the class C. Multiple inheritance is possible in AltaRica 3.0, although not recommended (as in all object-oriented languages).

The usage of the extends clause is illustrated in Figure 10.7. A generic class RepairableComponent is defined and then the class Pump extends this class RepairableComponent and add to it all its elements (state variables, events and the corresponding transitions).

```
domain RepairableState {WORKING, FAILED}

class RepairableComponent
    RepairableState s (init = WORKING);
    event failure, repair;
    transition
        failure: s == WORKING -> s := FAILED;
        repair: s == FAILED -> s := WORKING;
end

class Pump
    extends RepairableComponent;
    Boolean inFlow, outFlow (reset = false);
    assertion
        outFlow := s == WORKING and inFlow;
end
```

Figure 10.7: Illustration of inheritance usage.

This definition of the class Pump is equivalent to the one given Figure 10.8.

```
domain RepairableState {WORKING, FAILED}

class Pump
    RepairableState s (init = WORKING);
    Boolean inFlow, outFlow (reset = false);
    event failure, repair;
    transition
        failure: s == WORKING -> s := FAILED;
        repair: s == FAILED -> s := WORKING;
    assertion
        outFlow := s == WORKING and inFlow;
end
```

Figure 10.8: AltaRica 3.0 code of the class ‘Pump’.

From a technical point of view, there is less difference between composition and inheritance than it can appear at a first glance. When a class `A` (or a block `B`) extends a class `C` then the components of `C` are added to the class `A` (or to the block `B`) without any prefix. Thus, it is possible to reference the components of `C` in the class `A` (or in the block `B`) directly by their names. For example, in the class `Pump` that extends the class `RepairableComponent` (see Figure 10.8), the state variable `s` of the latter is referenced in the former by its name (without any prefix).

10.2.3 Aggregation (embeds-as clause)

Composition and inheritance are used to build hierarchical models organized in a tree. A block or a class can be composed of multiple blocks or instances of classes. However each block or each instance (of a class) is contained in only one block or in only one instance. Thus, a specific component, i.e. an object, cannot be contained into several blocks at the same time.

Safety studies take into account both physical and functional aspects of a system. In practice, the top event of a Fault Tree is almost always functional, e.g. “loss of the ability to provide something”. But the basic events of the Fault Tree are almost always failures of physical components. Also several undesirable events are often considered for the same system. Generally a component contributes to several functions and each function requires several components. So the question is how to create hierarchical models whose structure is not a tree but a Directed Acyclic Graph? i.e. how different branches of the hierarchy can share components?

In AltaRica 3.0, objects (i.e. blocks and instances of classes) may belong to different branches of a hierarchical model via the `embeds` clause. The use of this clause is illustrated by the (partial) code given Figure 10.9. The instance `T` of the class `Tank` is shared between the blocks `Line1` and `Line2`. Inside the block `Line1` (respectively `Line2`), the embedded object `T` is given an alias `T1` (respectively `T2`). Thus it must be referenced inside of the block `Line1` (respectively `Line2`) as `T1` (respectively `T2`).

```
block CoolingSystem
    Tank T;
    block Line1
        embeds main.T as T1;
        Pump P1;
        // ...
    end
    block Line2
        embeds main.T as T2;
        Pump P2;
        // ...
    end
    block Reactor
        // ...
    end
    // ...
end
```

Figure 10.9: Illustration of aggregation usage.

It is important to understand that only the prototype-oriented paradigm allows the definition of shared components. Indeed, a class defines an “on-the-shelf” component. So, a class cannot reference objects defined outside of this class. On the contrary, a block is always localized. It can refer to objects in the same model, i.e. in the block of the highest level (here the block `System`) or in the class within it is declared. A block declared inside a class cannot be referenced outside of this class.

10.2.4 Cloning (clones-as clause)

When designing models, it is usual to make copy of existing elements. For classes this notion is provided by the instantiation process whereas for blocks it is not the same. Conceptually, one wants to have the same notion of copy-paste files into a computer. The pasted file is a new file totally independent of the copied file. It starts its own life with the same elements as the copied file but any changes into it do not change the copied file. For blocks and in order to avoid duplicating the code, the solution is done by cloning.

In AltaRica 3.0, cloning is implemented by means of the `clones-as` clause. Code given Figure 10.10 illustrates this construct. A block `Line1` is declared and then, instead of copying the code of this block `Line1` for a new block `Line2`, the block `Line2` clones the block `Line1`.

```
block CoolingSystem
    Tank T;
    block Line1
        embeds owner.T as T;
        Pump P1;
        // ...
    end
    clones Line1 as Line2;
    block Reactor
        // ...
    end
    // ...
end
```

Figure 10.10: Illustration of cloning usage.

As previously introduced, block cloning is thus similar to class instantiation. It is more powerful however, because the cloned block may refer to external entities, while a class cannot refer anything to outside itself. Furthermore, one can remark the object `T` embedded into `Line1` is aliased with the same name `T` as its name.

10.2.5 Handling references to outside entities

Aggregation and cloning, coming from prototype-oriented paradigm, allow blocks to refer to external entities. Consider for instance, the (partial) model described Figure 10.11.

The block `S.B` embeds the block `S.A` referenced by `main.A` (an absolute path). Also the block `S.B.B2` clones the block `S.A.A1` referenced by `owner.owner.A.A1` (a relative path). These powerful mechanisms must be used carefully when connecting flow variables into assertions, specifically with cloned elements.

```
block S
    block A
        block A1
            // ...
        end
        // ...
    end

    block B
        embeds main.A as A
        // ...
        block B1
            block C
                // ...
            end
            // ...
        end
        block B2
            clones owner.owner.A.A1 as A1;
            // ...
        end
    end
    // ...
end
```

Figure 10.11: Cloning and external references.

10.2.6 Comparison and relations between classes and blocks

The comparison between classes and blocks is summarized in Figure 10.12. Relations between classes, instances (of classes) and blocks are represented in the diagram depicted Figure 10.13.

Concept	class	block
Definition	Generic component	Component having a unique occurrence in the model
Reuse	Instantiation and inheritance	Cloning and modifying
Usage	Multiple instances "On-the-shelf" component stored in a library	Unique occurrence "Sandbox"
Composition	Contains 'objects' ('instances' and 'blocks')	Contains 'objects' ('instances' and 'blocks')
Inheritance	Extends 'classes'	Extends 'classes'
Aggregation		Embeds 'objects' ('instances' and 'blocks')

Figure 10.12: classes vs. blocks.

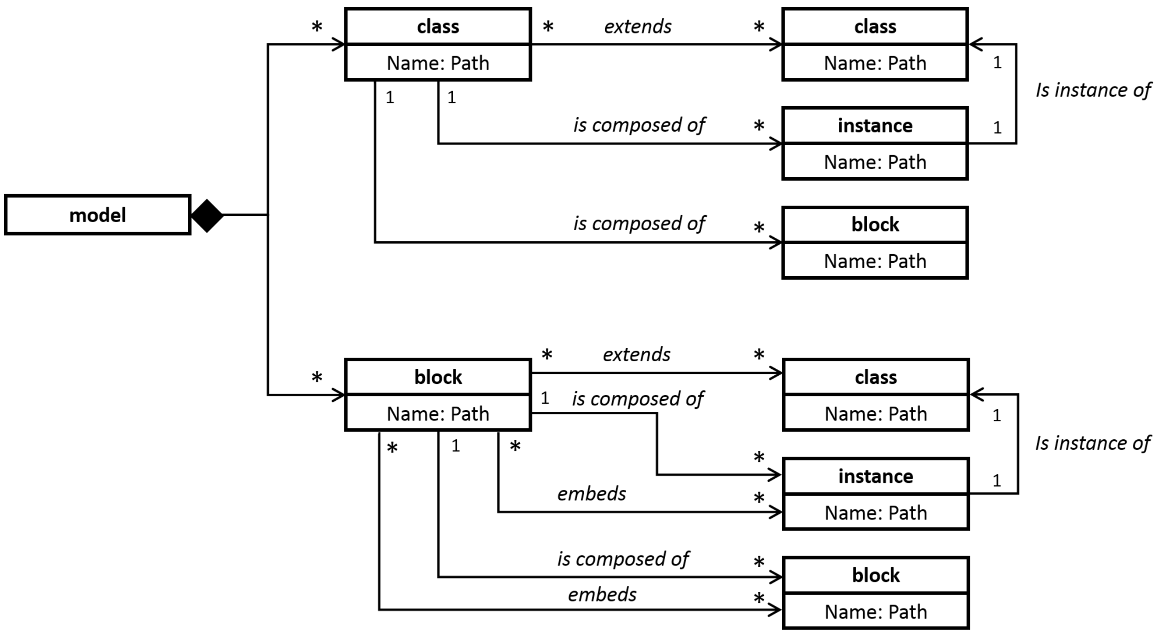


Figure 10.13: Relations between classes, objects and blocks.

10.3 Relations between components

Besides the “vertical” relations between components (composition, inheritance and aggregation), there are also their “horizontal” links, i.e. the means by which they interact. AltaRica 3.0 provides two mechanisms to represent interactions between components:

- Connections, i.e. the propagation of flows (of data, of matter, etc.) via the assertions, and
- Synchronizations of events.

10.3.1 Assertions

In the model of the Cooling system given Figure 10.9 all components are independent. To represent the propagation of the coolant from the tank to the reactor, we need to connect some flow variables together. To do that, several assertions are added to the previous model (see Figure 10.14)

```
block CoolingSystem
    Tank T;
    block Line1
        embeds owner.T as T1;
        Pump P1;
        assertion
            P1.inFlow := T1.outFlow;
    end
    block Line2
        embeds owner.T as T2;
        Pump P2;
        assertion
            P2.inFlow := T2.outFlow;
    end
    block Reactor
        Boolean inFlow (reset = false);
    end
    assertion
        Reactor.inFlow := Line1.P1.outFlow or Line2.P2.outFlow;
end
```

Figure 10.14: AltaRica 3.0 model of the Cooling system with assertions.

In AltaRica 3.0, there are two types of variables:

- State variables: they represent the state of components. They are identified by the attribute `init`, and optionally by the keyword `state`. The state variables are initialized once and for all at the beginning of the simulation. Their value is changed by the action of the fired transition. The `state` of the repairable component presented in Figure 10.7 is represented by a state variable `s`.
- Flow variables: they are used to model the propagation of the information, of the matter, of the energy, etc. between the components. They are identified by the attribute `reset`. Flow variables are recalculated after each transition firing via the assertions. The variables `inFlow` and `outFlow` in the class `Pump` in Figure 10.7 are flow variables. In the model given Figure 10.14 the input flow of the pump `P1` is connected to output flow of the tank `T`, the input flow of the reactor is connected to the output flows of the two pumps `P1` and `P2`, etc.

Assertions are presented in details in section 3.2.

10.3.2 Synchronizations

Another way to represent interactions between components is the synchronization of events. AltaRica 3.0 provides a powerful synchronization mechanism, which consists in compelling two or more events to occur simultaneously.

For example, to represent a common cause failure between the two pumps we need to add the new event `CCFPumps` and the corresponding transition in the block `CoolingSystem` (its AltaRica 3.0 code is given Figure 10.14). The new model is given Figure 10.15. The transition `CCFPumps` is fireable if at least one of the synchronized transitions is fireable.

```
block CoolingSystem
    // ...
    event CCFPumps;
    // ...
    transition
        CCFPumps: ?Line1.P1.failure & ?Line2.P2.failure
    // ...
end
```

Figure 10.15: AltaRica 3.0 model of the Cooling system with synchronizations.

Synchronizations are described in details in section 2.4.

10.4 Flattening rules

Each hierarchy of nested components can be flattened into a unique "flat" component, i.e. a component that does not contain any nested blocks and instances of classes, but only simple declarations and behavior clauses. This operation is called flattening.

Each hierarchical AltaRica 3.0 model can be flattened into a unique Guarded Transition System. Flattening of a hierarchical AltaRica 3.0 model is a purely syntactic operation. It works in three steps:

1. Flattening of the hierarchy,
2. Flattening of the synchronizations,
3. Hiding.

In the following we will make a distinction between variables representing instances of classes and variables representing states and flows. We will name atomic variables these state and flow variables.

Finally, the domains are also included into the GTS model for they are required to type atomic variables taking their values into them.

10.4.1 Flattening of the hierarchy

Classes and blocks are flattened in a similar way. However, there are some differences due to the fact that the structure of classes and blocks is not the same.

Class flattening

Consider a class $C = \langle C_E, B_A, D, T, A \rangle$ composed of:

- a set of extended classes C_E;
- a set of declared elements D, including instances of classes and atomic elements, i.e. atomic variables, events, parameters and observers;
- a set of declared blocks B_A;
- a behavior clause, including a set of transitions T and a set of assertions A.

A class C_{flat} is a flat form of the class C. It is obtained in the following way:

1. First of all an empty class C_{flat} is created.
2. Second, the flattening of extended classes from C_E is performed:
    - Each class K from C_E (i.e. such that C extends K) is flattened into a class K_{flat}.
    - For each class K, a copy (without any prefix) of K_{flat} is added to C_{flat}.
3. Third, declared blocks B_A are flattened:
    - Each block b such that C is composed of b (i.e. $b \in B_A$) is flattened into a block b_{flat}.
    - For each block b, the block b_{flat} is added to C_{flat}.
4. Then, declared instances of classes from D are flattened:
    - Each class Q such that C is composed of an instance q of Q (i.e. $q \in D$) is flattened into a class Q_{flat}.
    - For each instance q of a class Q, a copy of the class Q_{flat} is added to C_{flat}. During the copy, all named atomic elements (atomic variables, events, parameters, observers and their references in expressions and synchronizations) are prefixed with the name of the instance followed by a dot.
5. Next, atomic variable declarations, event declarations, parameter declarations and observer declarations of C are copied to C_{flat}. By atomic variable we mean variable whose type is either a basic type (Boolean, Integer, Real, Symbol or a user defined domain).
6. Finally, the behavior clause is flattened:
    - Transitions of C are copied to C_{flat}.
    - Assertions of C are copied to C_{flat}.

Block flattening

Consider a block $B = \langle C_E, B_A, B_E, D, T, A \rangle$ composed of:

- a set of extended classes C_E,
- a set of declared blocks B_A,
- a set of embedded objects (blocks and instances of classes) B_E,
- a set of declared elements D, including instances of classes and atomic elements, i.e. atomic variables, events, parameters and observers,
- a behavior clause, including a set of transitions T and a set of assertions A.

A block B_{flat} is a flat form of block B. It is obtained in the following way:

1. First, an empty block B_{flat} is created.
2. Second, extended classes are flattened:
    - Each class K from C_E (i.e. such that B extends K) is flattened into a class K_{flat}.
    - For each class K, a copy (without any prefix) of K_{flat} is added to B_{flat}.
3. Third, declared blocks from B_A are flattened:

- Each block b such that B is composed of b (i.e. $b \in B_A$) is flattened into a block b_{flat}.
- For each block b, b_{flat} is added to B_{flat}.

4. Then, declared elements of D are flattened:
   - Each class Q such that B is composed of an instance q of Q (i.e. $q \in D$) is flattened into a class Q_{flat}.
   - For each instance q of a class Q, a copy of the class Q_{flat} is added to the block B_{flat}. During the copy, all named atomic elements (atomic variables, events, parameters, observers and their references in expressions and synchronizations) are prefixed with the name of the instance followed by a dot.
5. Next, atomic variable declarations, event declarations, parameter declarations and observer declarations of B are copied to B_{flat}. During the copy they are prefixed by the name of the block followed by a dot. References of variables and parameters in expressions are also prefixed by the name of the block followed by a dot, except for those that belong to embedded objects (blocks and instances of classes).
6. Finally, the behavior clause is flattened:
   - Transitions of B are copied to B_{flat}.
   - Assertions of B are copied to B_{flat}
   - During the copy, references to atomic variables in expressions are prefixed by the name of the block followed by a dot. References to atomic variables and events belonging to embedded objects are not prefixed. Aliases of embedded objects are replaced by their paths.

Example

Consider the model of the Cooling system given Figure 10.14. Figure 10.16 illustrates how the algorithm of hierarchy flattening works by presenting the flattened Cooling System corresponding to the model given Figure 10.14.

```
domain RepairableState {WORKING, FAILED}

block CoolingSystem
    Boolean T.isEmpty (init = false);
    Boolean T.outFlow (reset = true);
    RepairableState Line1.P1.s (init = WORKING);
    RepairableState Line2.P2.s (init = WORKING);
    Boolean Reactor.inFlow (reset = false);
    Boolean Line1.P1.inFlow (reset = false);
    Boolean Line2.P2.inFlow (reset = false);
    Boolean Line1.P1.outFlow (reset = false);
    Boolean Line2.P2.outFlow (reset = false);
    event T.getEmpty;
    event Line1.P1.repair;
    event Line2.P2.repair;
    event Line1.P1.failure;
    event Line2.P2.failure;
    transition
        Line1.P1.failure: Line1.P1.s == WORKING -> Line1.P1.s := FAILED;
        Line1.P1.repair: Line1.P1.s == FAILED -> Line1.P1.s := WORKING;
        Line2.P2.failure: Line2.P2.s == WORKING -> Line2.P2.s := FAILED;
        Line2.P2.repair: Line2.P2.s == FAILED -> Line2.P2.s := WORKING;
        T.getEmpty: not T.isEmpty -> T.isEmpty := true;
    assertion
        Line1.P1.outFlow := Line1.P1.s == WORKING and Line1.P1.inFlow;
        Line1.P1.inFlow := T.outFlow;
        Line2.P2.outFlow := Line2.P2.s == WORKING and Line2.P2.inFlow;
        Line2.P2.inFlow := T.outFlow;
        T.outFlow := not T.isEmpty;
        Reactor.inFlow := Line1.P1.outFlow or Line2.P2.outFlow;
end
```

Figure 10.16: Flattened Cooling System.

10.4.2 Flattening of the synchronizations

Transitions of the class C (or the block B) are in the following form:

$$e : !a_1 \; \& \; ... \; \& \; !a_m \; \& \; ?b_1 \; \& \; ... \; \& \; ?b_n \; \& \; L_1 \rightarrow R_1 \; \& \; ... \; \& \; L_r \rightarrow R_r$$
$$m \geq 0, n \geq 0, r \geq 0$$

where:

1. $e, a_i \; (i = 0...m)$ and $b_j \; (j = 0...n)$ are the events;
2. Events are prefixed by either ! or ?, called the modality: ! meaning that the event is mandatory and ? meaning that the event is optional;
3. $L_k \; (k = 0...r)$ are Boolean expressions and $R_k \; (k = 0...r)$ are instructions.

They are flattened in the following way:

First case: $m \geq 1$ or $r \geq 1$

For each set of transitions (there may be several):

$$a_1 : \; G_1 \rightarrow P_1, ..., a_m : \; G_m \rightarrow P_m$$

$$b_1: \; H_1 \rightarrow Q_1, ..., b_n: \; H_n \rightarrow Q_n \; (n \geq 0)$$

the following new transition is created:

$$\begin{gathered} e: \; G_1 \; and \; ... \; and \; G_m \; and \; L_1 \; and \; ... \; and \; L_r \\ \rightarrow \{P_1; ...; P_m; if \; H_1 \; then \; Q_1; ...; if \; H_n \; then \; Q_n; R_1; ...; R_r\} \end{gathered}$$

The modality ! forces the corresponding synchronized transition to be fireable.

Second case: $m = 0$ and $r = 0$

1. $n > 1$:
   For each set of transitions (there may be several):
   $$b_1: \; H_1 \rightarrow Q_1, ..., \; b_n: \; H_n \rightarrow Q_n$$
   the following new transition is created:
   $$e: \; H_1 \; or \; ... \; or \; H_n \rightarrow \{if \; H_1 \; then \; Q_1; ...; if \; H_n \; then \; Q_n\}$$
2. $n = 1$
   For each set of transitions (there may be several):
   $$b_1: \; H_1 \rightarrow Q_1$$
   the following new transition is created:
   $$e: \; true \rightarrow \{if \; H_1 \; then \; Q_1\}$$

In other words, the synchronizing transition is fireable if at least one of the synchronized transitions is. The action of the synchronizing transition consists in firing all fireable synchronized transitions.

10.4.3 Hiding

Events involved in a synchronization continue to exist individually. However, if they must not occur individually, they can be hidden using the hidden mechanism: set the attribute `hidden`, of the events to hide, to the value true. By default this attribute is set to the value false, meaning the event is not hidden. As for all others attributes, it can be overloaded at instantiation. Finally, hidden events and the transitions they label are just removed from C_{flat} or from B_{flat}.

Chapter 11

Stochastic Models

The semantics of Stochastic Guarded Transition Systems has been defined chapter 3. Recall that a Stochastic Guarded Transition System is tuple $\langle V = S \cup F, E, T, A, \iota, \mathit{delay}, \mathit{expectation}, \mathit{policy}\rangle$ where:

- $\langle V = S \cup F, E, T, A, \iota\rangle$ is a GTS;
- *delay* is a function from events and oracles to positive real numbers.
- *expectation* is a function from events to positive real numbers.
- *policy* is a function from events to $\{\mathit{restart}, \mathit{memory}\}$.

The three functions *delay*, *expectation* and *policy* are defined by means of attributes associated with events. Functions *expectation* and *policy*, previously introduced in part. 2.7 are simple and deserve no further explanation. This chapter is devoted to the function *delay* which is more complex and involves specific constructs.

11.1 Role of the function "delay"

The function "delay" should be seen essentially as the inverse of an (invertible) cumulative probability distribution. A cumulative probability distribution is a monotone non decreasing function from positive real numbers (representing the time) to the real range $[0, 1]$, as illustrated Figure 11.1. It associates with each time t the probability that a given event occurs before time t.

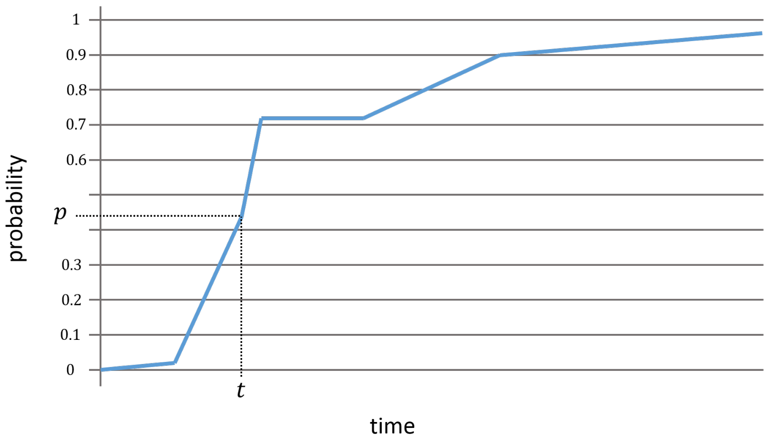


Figure 11.1: A Cumulative Probability Distribution.

Therefore, given a probability p (obtained through the oracle), the "delay" function returns the first (the earliest) time t such that the given event has a probability p to occur before time t.

In practice, the limit as the time goes to infinity of the Cumulative Probability Distribution may be less than 1. In that case, the delay function returns $+\infty$ for probabilities greater than the limit (next section will illustrate this point).

11.2 Built-in Delay Functions

AltaRica 3.0 implements a few built-in delay functions and a generic mechanism to defined delay functions point wisely. The built-in functions are the most widely used in Reliability analyses.

11.2.1 Dirac Delays

A Dirac delay is a deterministic delay: the event occurs after a fixed delay d, as illustrated Figure 11.2. The formal definition of this delay is as follows:

$$probability(t, Dirac(d)) = \begin{cases} 0, & if\ t < d \\ 1, & otherwise \end{cases}$$

$$delay(p, Dirac(d)) = d$$

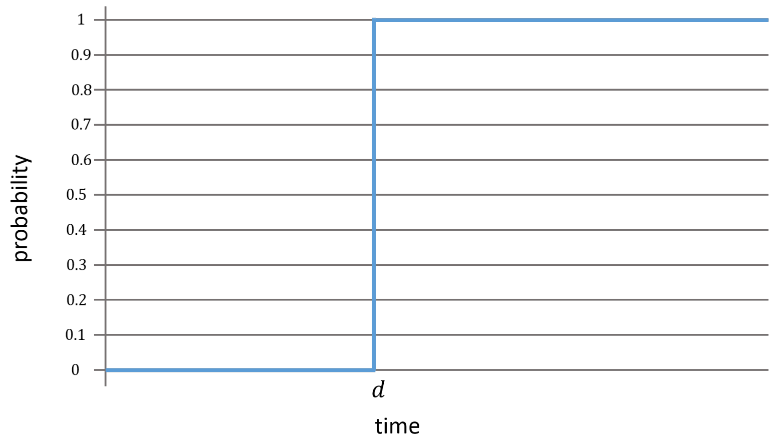


Figure 11.2: Dirac Delay.

The AltaRica 3.0 code to model Dirac delay consists just to put d as the value of the attribute delay, as illustrated below:

```
    // ...
    parameter Real d = 72.0;
    event maintenance (delay = Dirac(d));
    // ...
```

11.2.2 Constant Delays

Another basic type of cumulative probability distribution is the constant distribution which corresponds to the point wise probability in Fault Trees: the event occurs at time 0 with a probability q, as illustrated Figure 11.3. The formal definition of the constant delay is as follows:

$$probability(t, constant(q)) = q$$

$$delay(p, constant(q)) = \begin{cases} 0, \ if \ p \leq q \\ +\infty, \ otherwise \end{cases}$$

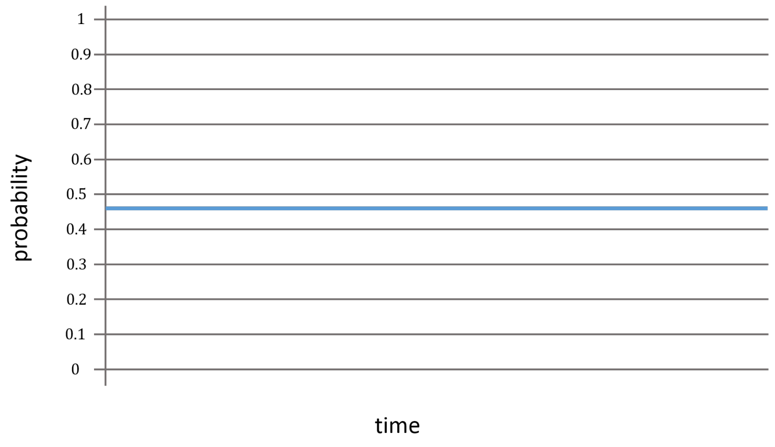


Figure 11.3: Constant Delay.

This type of delay is obtained by means of the built-in “constant” as illustrated by the AltaRica 3.0 code below:

```
    // ...
    parameter Real p = 0.38;
    event failure (delay = constant(p));
    // ...
```

11.2.3 Exponential Delays

The most widely used probability distribution is the exponential one. This document contains many example of this distribution, which is illustrated Figure 11.4 and introduced by the built-in “exponential”. The formal definition of this delay is as follows:

$$probability(t, exponential(\lambda)) = 1 - exp(-\lambda \times t)$$
$$delay(p, exponential(\lambda)) = -\frac{log(1-p)}{\lambda}$$

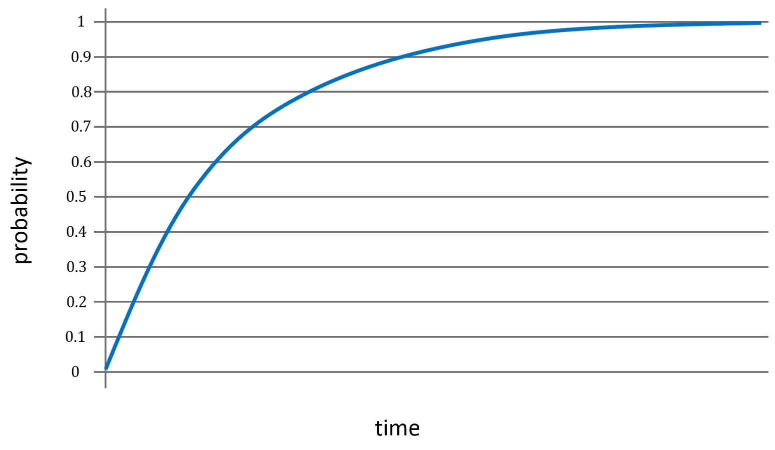


Figure 11.4: Exponential Delay.

The AltaRica 3.0 code for an exponential delay is illustrated below:

```
    // ...
    parameter Real lambda = 0.38;
    event failure (delay = exponential(lambda));
    // ...
```

11.2.4 Weibull Delays

Another frequently used cumulative distribution function is the Weibull distribution, as illustrated Figure 11.5. It takes two parameters: a shape factor β and a scale factor α. It is formally defined as follows:

$$probability(t, Weibull(\alpha, \beta)) = 1 - exp(-(\frac{t}{\alpha})^{\beta})$$
$$delay(p, Weibull(\alpha, \beta)) = \alpha \times (-log(1-p))^{\frac{1}{\beta}}$$

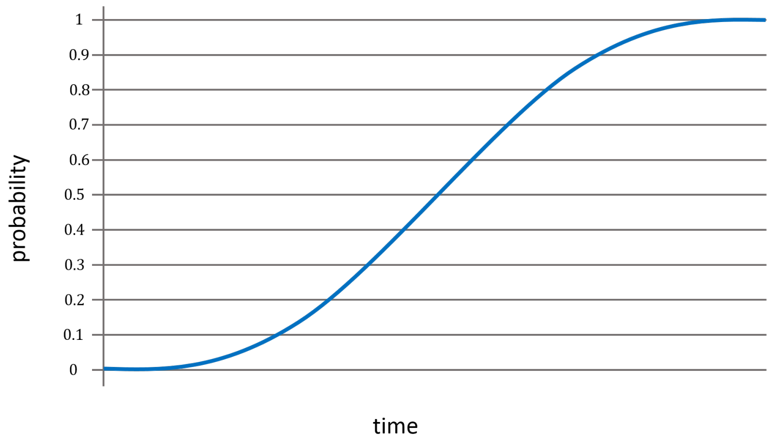


Figure 11.5: Weibull Delay.

The AltaRica 3.0 code for a Weibull delay is illustrated below:

```
    // ...
    parameter Real alpha = 1.0e3;
    parameter Real beta = 2;
    event failure (delay = Weibull(alpha,beta));
    // ...
```

11.2.5 Uniform Delays

AltaRica 3.0 also implements the uniform distribution function, as illustrated Figure 11.6. It takes two parameters, also called boundaries, a and b. It is formally defined as follows:

$$probability(t, uniform(a,b)) = \begin{cases} 0, \ if \ t < a \\ \frac{t-a}{b-a}, \ if \ a \leq t < b \\ 1, \ if \ t \leq b \end{cases}$$

$$delay(p, uniform(a,b)) = a + (b-a) \times p$$

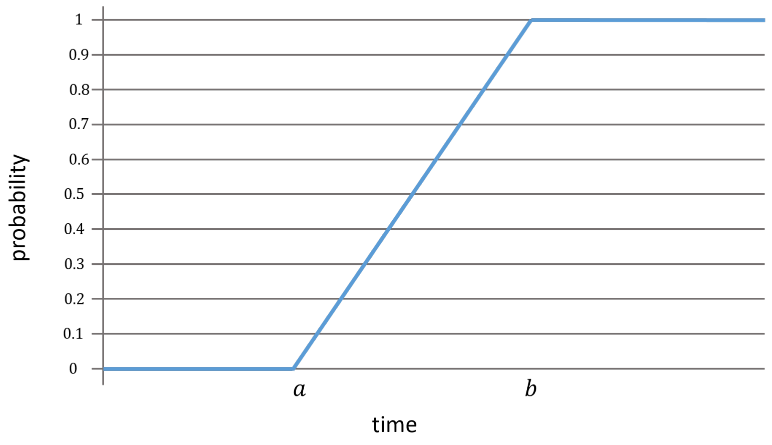


Figure 11.6: Uniform Delay.

The AltaRica 3.0 code for a Weibull delay is illustrated below:

```
    // ...
    parameter Real a = 2;
    parameter Real b = 10;
    event failure (delay = uniform(a,b));
    // ...
```

11.2.6 User Defined Curves

AltaRica 3.0 provides a generic mechanism to describe any cumulative distribution function by means of a set of points interpolated in a triangular way, as illustrated Figure 11.7. The distribution is given as an ordered list of points $[t_0 : p_0, t_1 : p_1, ..., t_n : p_n]$ such that:

- $0 = t_0 \leq t_1 \leq ... \leq t_n$
- $p_0 \leq p_1 \leq ... \leq p_n$

The probability and the delay are calculated as follows:

$$probability(t, [t_0 : p_0, t_1 : p_1, ..., t_n : p_n]) = \begin{cases} if\ \exists i \in [0; n[\ /\ t \in [t_i; t_{i+1}[,\ p_i + (p_{i+1} - p_i) \times \frac{t - t_i}{t_{i+1} - t_i} \\ else\ p_n \end{cases}$$

$$delay(p, [t_0 : p_0, t_1 : p_1, ..., t_n : p_n]) = \begin{cases} if\ \exists i \in [0; n[\ /\ p \in [p_i; p_{i+1}[,\ t_i + (t_{i+1} - t_i) \times \frac{p - p_i}{p_{i+1} - p_i} \\ else\ \ +\infty \end{cases}$$

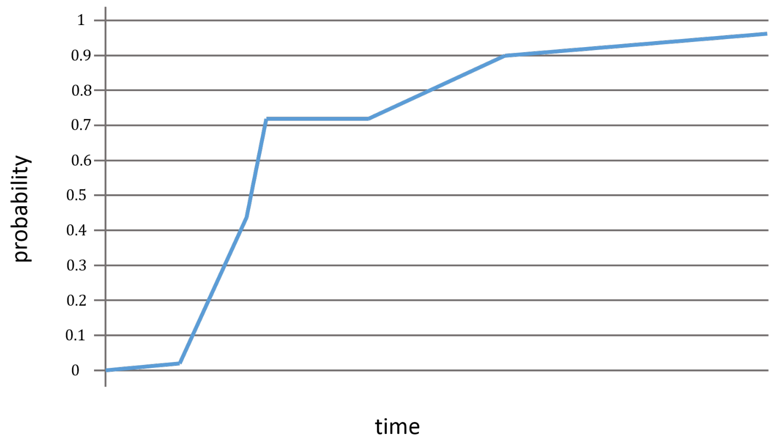


Figure 11.7: User Defined Cumulative Distribution Function.

The AltaRica 3.0 code for a user defined probability distribution is illustrated below:

```
    // ...
    event failure (delay = [ 0:0.0, 876:0.2, 1752:0.4, 2628:0.45, 3504:0.5,
                             4380:0.5, 5256:0.55, 6132:0.6, 7446:0.6, 8760:0.6 ]);
    // ...
```

The syntax for curves is as follows:

```
Curve ::=
      "[" Real ":" Real ( "," Real ":" Real )* "]"
```

Appendices

Appendix A

AltaRica 3.0 E-BNF

A.1 Comments

```
Comment ::=
      PARAGRAPH_COMMENT | LINE_COMMENT

PARAGRAPH_COMMENT ::=
      "/*" .* "*/"

LINE_COMMENT ::=
      "//" .* "\n"
```

A.2 Identifiers and Paths

```
ALPHA ::=
      "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k"
    | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v"
    | "w" | "x" | "y" | "z"
    | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K"
    | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V"
    | "W" | "X" | "Y" | "Z"

DIGIT ::=
      "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

Identifier ::=
      ( ALPHA | "_" ) ( DIGIT | ALPHA | "_" )*

Path ::=
      Identifier ( "." Path )?
    | "owner" "." Path
    | "main" "." Path
    | Identifier "::" Path
```

A.3 Literals

```
Boolean ::=
      "false" | "true"

Integer ::=
      DIGIT+
// Negative integers are defined thanks to the minus operator of expressions

Mantissa ::=
      ( "e" | "E" ) ( "+" | "-" )? DIGIT+

Real ::=
      DIGIT+ ( "." DIGIT+ )? Mantissa
// Negative reals are defined thanks to the minus operator of expressions

LiteralValue ::=
      Boolean | Integer | Real
    | LiteralAttributeValue
    | SymbolicConstant

LiteralAttributeValue ::=
      "restart" | "memory"

SymbolicConstant ::=
      Identifier
```

A.4 Types

```
Type ::=
      BaseType
    | TypeReference

BaseType ::=
      "Boolean" | "Integer" | "Real" | "Symbol"

TypeReference ::=
      DomainReference
    | RecordReference
    | ClassReference

DomainReference ::=
      Path

RecordReference ::=
      Path

ClassReference ::=
      Path
```

A.5 Expressions

```
Expression ::=
      "(" Expression ")"
    | Expression NOperator Expression
    | UOperator Expression
    | OperatorCall
    | IfThenElseExpression
    | SwitchExpression
    | Curve
    | LiteralExpression

NOperator ::=
      "or" | "and"
    | "==" | "!=" | "<" | "<=" | ">" | ">="
    | "+" | "-" | "*" | "/"

UOperator ::=
      "not" | "-" | "+"

OperatorCall ::=
      OperatorName "(" ( Expression ( "," Expression )* )? ")"

OperatorName ::=
      Path
    | BuiltinName
    | DelayName

BuiltinName ::=
      "min" | "max" | "abs" | "ceil" | "exp" | "floor"
    | "log" | "log10" | "mod" | "pow" | "sqrt" | "#"
    | "Real" | "Integer"

DelayName ::=
      "Dirac" | "exponential" | "Weibull" | "constant" | "uniform"

IfThenElseExpression ::=
      "if" Expression "then" Expression "else" Expression

SwitchExpression ::=
      "switch" "{"
        ( "case" Expression ":" Expression )*
        "default" ":" Expression
      "}"
```

```
Curve ::=
      "[" Real ":" Real ( "," Real ":" Real )* "]"

LiteralExpression ::=
      VariableReference
    | ParameterReference
    | ObserverReference
    | LiteralValue

VariableReference ::=
      Path

ParameterReference ::=
      Path

ObserverReference ::=
      Path
```

A.6 Instructions

```
Instruction ::=
      "skip" ";"
    | VariableReference ":=" Expression ";"
    | VariableReference ":=:" VariableReference ";"
    | "if" Expression "then" Instruction ( "else" Instruction )?
    | "switch" "{"
        ( "case" Expression ":" Instruction )*
        "default" ":" Instruction
      "}"
    | "{" ( Instruction )+ "}"
    | FunctionCall

FunctionCall ::=
      FunctionName "(" ( Expression ( "," Expression )* )? ")"

FunctionName ::=
      Path
```

A.7 Blocks and Classes

```
BlockDeclaration ::=
      "block" Path
        ( BlockClause )*
      "end"

ClassDeclaration ::=
      "class" Path
        ( ClassClause )*
      "end"

BlockClause ::=
      BlockDeclarationClause
    | TransitionsClause
    | AssertionClause

ClassClause ::=
      ClassDeclarationClause
    | TransitionsClause
    | AssertionClause

BlockDeclarationClause ::=
      ClassDeclarationClause
    | EmbedsClause
    | ClonesClause

ClassDeclarationClause ::=
      ExtendsClause
    | VariableDeclaration
    | StateDeclaration
    | EventDeclaration
    | ParameterDeclaration
    | ObserverDeclaration
    | BlockDeclaration
```

```
EmbedsClause ::=
      "embeds" Path "as" Identifier ";"

ClonesClause ::=
      "clones" Path "as" Identifier Attributes? ";"

ExtendsClause ::=
      "extends" Path Attributes? ";"
```

```
VariableDeclaration ::=
      Type Path ( "," Identifier )* Attributes? ";"

StateDeclaration ::=
      "state" Type Path ( "," Identifier )* Attributes? ";"

EventDeclaration ::=
      "event" Path ( "," Identifier )* Attributes? ";"

ParameterDeclaration ::=
      "parameter" Type Path "=" Expression ";"

ObserverDeclaration ::=
      "observer" Type Path "=" Expression ";"
```

```
TranstionsClause ::=
      "transition" TranstionsDeclaration+

TransitionDeclaration:
      Path ":" Transition ( "|" Transition )*

Transition ::=
      IndividualTransition ( "&" IndividualTransition )*

IndividualTransition ::=
      Expression "->" Instruction
    | SynchronizedEvent

SynchronizedEvent ::=
      "!" Path
    | "?" Path

AssertionClause ::=
      "assertion" Instruction+
```

A.8 Attributes

```
Attributes ::=
      "(" Attribute ( "," Attribute )* ")"

Attribute ::=
      AttributeType "=" Expression

AttributeType ::=
      Path
    | ( Path "." )? AtomicAttributeType

AtomicAttributeType ::=
      "init" | "reset"
    | "delay" | "expectation" | "policy" | "hidden"
```

A.9 Preprocessed Elements

```
IncludeDirective ::=
      "include" IncludeSequence

IncludeSequence ::=
      """ ( . -( \" \< \> \  ) ) ( . -( \" \< \> ) )*
        ( . -( \" \< \> \  ) ) """

ConstantDeclaration ::=
      "constant" Type Path "=" Expression ";"

RecordDeclaration ::=
      "record" Path
        RecordDeclarationClause+
      "end"

RecordDeclarationClause ::=
      ExtendsClause | VariableDeclaration

ExtendsClause ::=
      "extends" Path Attributes? ";"

VariableDeclaration ::=
      Type Path ( "," Identifier )* Attributes? ";"

FunctionDeclaration ::=
      "function" Path "(" Arguments? ")"
        Instruction+
      "end"

OperatorDeclaration ::=
      "operator" Type Path "(" Arguments? ")"
        Expression
      "end"

Arguments ::=
      Argument ( ","Argument )*

Argument ::=
      Type Identifier
```

A.10 Domains

```
DomainDeclaration ::=
      "domain" Path "{" SymbolicConstants "}"

SymbolicConstants ::=
      SymbolicConstant ( "," SymbolicConstant )*

SymbolicConstant ::=
      Identifier
```

A.11 AltaRica 3.0 model

```
AltaRica3Model ::=
      ModelDeclaration*

ModelDeclaration ::=
      IncludeDirective
    | ConstantDeclaration
    | DomainDeclaration
    | RecordDeclaration
    | FunctionDeclaration
    | OperatorDeclaration
    | ClassDeclaration
    | BlockDeclaration
```

Appendix B

GTS Grammar

B.1 E-BNF grammar

B.1.1 Identifiers and Paths

```
ALPHA ::=
      "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k"
    | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v"
    | "w" | "x" | "y" | "z"
    | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K"
    | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V"
    | "W" | "X" | "Y" | "Z"

DIGIT ::=
      "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

Identifier ::=
      ( ALPHA | "_" ) ( DIGIT | ALPHA | "_" )*

Path ::=
      Identifier ( ( "." | "::" ) Path ) ?
```

B.1.2 Constant values

```
Boolean ::=
      "false" | "true"

Integer ::=
      DIGIT+
// Negative integers are defined thanks to the minus operator of expressions

Real ::=
      DIGIT+ ( "." DIGIT+ )? ( ( "e" | "E" ) ( "+" | "-" )? DIGIT+ )?
// Negative reals are defined thanks to the minus operator of expressions

LiteralValue ::=
      Boolean | Integer | Real
    | LiteralAttributeValue
    | SymbolicConstant

LiteralAttributeValue ::=
      "restart" | "memory"

SymbolicConstant ::=
      Identifier
```

B.1.3 Types

```
Type ::=
      BaseType
    | DomainReference

BaseType ::=
      "Boolean" | "Integer" | "Real" | "Symbol"

DomainReference ::=
      Path
```

B.1.4 Expressions

```
Expression ::=
      "(" Expression ")"
    | "if" Expression "then" Expression "else" Expression
    | Expression NOperator Expression
    | UOperator Expression
    | OperatorCall "(" ( Expression ( "," Expression )* )? ")"
    | Curve
    | LiteralExpression

NOperator ::=
      "or" | "and"
    | "==" | "!=" | "<" | "<=" | ">" | ">="
    | "+" | "-" | "*" | "/"

UOperator ::=
      "not" | "+" | "-"

OperatorCall ::=
      "min" | "max" | "abs" | "ceil" | "exp" | "floor"
    | "log" | "log10" | "mod" | "pow" | "sqrt" | "#"
    | "Dirac" | "exponential" | "Weibull" | "constant" | "uniform"
    | "Real"|  "Integer"
    | Path

Curve ::=
      "[" Real ":" Real ( "," Real ":" Real )* "]"

LiteralExpression ::=
      VariableReference
    | ParameterReference
    | ObserverReference
    | LiteralValue

VariableReference ::=
      Path

ParameterReference ::=
      Path

ObserverReference ::=
      Path
```

B.1.5 Instructions and Assertion

```
Instruction ::=
      "skip" ";"
    | VariableReference ":=" Expression ";"
    | "if" Expression "then" VariableReference ":=" Expression ";"
    | "{" ( Instruction )+ "}"

AssertionDeclaration ::=
      "assertion" Instruction+
```

B.1.6 Attributes

```
Attribute ::=
      AttributeType "=" Expression

AttributeType ::=
      Path
    | ( Path "." )? AtomicAttributeType

AtomicAttributeType ::=
      "init" | "reset"
    | "delay" | "expectation" | "policy" | "hidden"
```

B.1.7 Domains

```
DomainDeclaration ::=
      "domain" Path "{" SymbolicConstants "}"

SymbolicConstants ::=
      SymbolicConstant ( "," SymbolicConstant )*
```

B.1.8 Element Declarations

```
VariableDeclaration ::=
      Type Path Attributes? ";"

ParameterDeclaration ::=
      Type Path "=" Expression ";"

EventDeclaration ::=
      "event" Path Attributes? ";"

ObserverDeclaration ::=
      "observer" Type Path "=" Expression ";"

Attributes ::=
      "(" Attribute ( "," Attribute )* ")"
```

B.1.9 Transitions

```
TransitionDeclaration ::=
      "transition" Transition+

Transition ::=
      EventReference ":" Expression "->" Instruction

EventReference ::=
      Path
```

B.1.10 Guarded Transition Systems

```
GTS ::=
      Declaration*

Declaration ::=
      DomainDeclaration
    | GTSModelDeclaration

GTSModelDeclaration ::=
      "block" Path GTSDeclaration* "end"

GTSDeclaration ::=
      VariableDeclaration
    | ParameterDeclaration
    | ObserverDeclaration
    | EventDeclaration
    | TransitionDeclaration
    | AssertionDeclaration
```

B.2 XML grammar

The following defines the XSD (XML Schema Definition) of the GTS grammar. It begin with the following rules

```
<xs:element name="altarica">
  <xs:complexType>
    <xs:group ref="Definition" minOccurs="0" maxOccurs="unbounded"/>
  </xs:complexType>
</xs:element>

<xs:group name="Definition">
  <xs:choice>
    <xs:element ref="define-domain"/>
    <xs:element ref="define-block"/>
  </xs:choice>
</xs:group>
```

B.2.1 Domain definition

```
<xs:element name="define-domain">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Symbol" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

B.2.2 Block definition

```
<xs:element name="define-block">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="block-body" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

<xs:group name="block-body">
  <xs:choice>
    <xs:element ref="define-variable"/>
    <xs:element ref="define-parameter"/>
    <xs:element ref="define-observer"/>
    <xs:element ref="define-event"/>
    <xs:element ref="transition"/>
    <xs:element ref="assertion"/>
  </xs:choice>
</xs:group>
```

```
<xs:element name="define-variable">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="attribute" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="type" type="type-definition" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="define-parameter">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="expression" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="type" type="type-definition" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="define-observer">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="expression" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="type" type="type-definition" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="define-event">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="attribute" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="transition">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="event"/>
      <xs:group ref="expression" minOccurs="1" maxOccurs="1"/>
      <xs:group ref="instruction-group" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="assertion">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="instruction-group" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

B.2.3 Instructions

```xml
<xs:group name="instruction-group">
  <xs:choice>
    <xs:element ref="assignment"/>
    <xs:element ref="conditional-assignment"/>
    <xs:element ref="block"/>
    <xs:element ref="skip"/>
  </xs:choice>
</xs:group>

<xs:element name="assignment">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="variable" minOccurs="1" maxOccurs="1"/>
      <xs:group ref="expression" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="conditional-assignment">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="expression" minOccurs="1" maxOccurs="1"/>
      <xs:element ref="variable" minOccurs="1" maxOccurs="1"/>
      <xs:group ref="expression" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="block">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="instruction-group" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="skip">
</xs:element>
```

B.2.4 Expressions

```
<xs:group name="expression">
  <xs:choice>
    <xs:element ref="curve"/>
    <xs:element ref="ite"/>
  <xs:group ref="relational-operator-expression"/>
    <xs:group ref="logical-operator-expression"/>
  <xs:group ref="arithmetic-operator-expression"/>
  <xs:group ref="builtin-operator-expression"/>
    <xs:group ref="builtin-expression"/>
    <xs:group ref="reference-element"/>
    <xs:group ref="literal"/>
  </xs:choice>
</xs:group>
```

```
<xs:element name="curve">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="point" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="point">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Real" minOccurs="2" maxOccurs="2"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
<xs:element name="ite">
  <xs:complexType>
    <xs:group ref="expression" minOccurs="3" maxOccurs="3" />
  </xs:complexType>
</xs:element>
```

```
<xs:group name="relational-operator-expression">
  <xs:choice>
    <xs:element name="gt">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="2" maxOccurs="2" />
      </xs:complexType>
    </xs:element>
    <xs:element name="lt">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="2" maxOccurs="2" />
      </xs:complexType>
    </xs:element>
    <xs:element name="geq">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="2" maxOccurs="2" />
      </xs:complexType>
    </xs:element>
    <xs:element name="leq">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="2" maxOccurs="2" />
      </xs:complexType>
    </xs:element>
    <xs:element name="eq">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="2" maxOccurs="2" />
      </xs:complexType>
    </xs:element>
    <xs:element name="dif">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="2" maxOccurs="2" />
      </xs:complexType>
    </xs:element>
  </xs:choice>
</xs:group>
```

```
<xs:group name="logical-operator-expression">
  <xs:choice>
    <xs:element name="not">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="1" />
      </xs:complexType>
    </xs:element>
    <xs:element name="or">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="unbounded" />
      </xs:complexType>
    </xs:element>
    <xs:element name="and">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="unbounded" />
      </xs:complexType>
    </xs:element>
  </xs:choice>
</xs:group>
```

```
<xs:group name="arithmetic-operator-expression">
  <xs:choice>
    <xs:element name="neg">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="1" />
      </xs:complexType>
    </xs:element>
    <xs:element name="add">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="unbounded" />
      </xs:complexType>
    </xs:element>
    <xs:element name="sub">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="unbounded" />
      </xs:complexType>
    </xs:element>
    <xs:element name="mul">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="unbounded" />
      </xs:complexType>
    </xs:element>
    <xs:element name="div">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="unbounded" />
      </xs:complexType>
    </xs:element>
    <xs:element name="plus">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="unbounded" />
      </xs:complexType>
    </xs:element>
    <xs:element name="minus">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="unbounded" />
      </xs:complexType>
    </xs:element>
  </xs:choice>
</xs:group>
```

```
<xs:group name="builtin-operator-expression">
  <xs:choice>
    <xs:element name="min">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="unbounded" />
      </xs:complexType>
    </xs:element>
    <xs:element name="max">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="unbounded" />
      </xs:complexType>
    </xs:element>
    <xs:element name="abs">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="1" />
      </xs:complexType>
    </xs:element>
    <xs:element name="ceil">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="1" />
      </xs:complexType>
    </xs:element>
    <xs:element name="exp">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="1" />
      </xs:complexType>
    </xs:element>
    <!-- ... -->
    <!-- ... -->
    <xs:element name="floor">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="1" />
      </xs:complexType>
    </xs:element>
    <xs:element name="log">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="1" />
      </xs:complexType>
    </xs:element>
    <xs:element name="log10">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="1" />
      </xs:complexType>
    </xs:element>
    <xs:element name="mod">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="2" maxOccurs="2" />
      </xs:complexType>
    </xs:element>
    <xs:element name="pow">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="2" maxOccurs="2" />
      </xs:complexType>
    </xs:element>
    <xs:element name="sqrt">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="1" />
      </xs:complexType>
    </xs:element>
    <xs:element name="count">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="unbounded" />
      </xs:complexType>
    </xs:element>
  </xs:choice>
</xs:group>
```

```
<xs:group name="builtin-expression">
  <xs:choice>
    <xs:element name="Dirac">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="1" />
      </xs:complexType>
    </xs:element>
    <xs:element name="exponential">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="1" />
      </xs:complexType>
    </xs:element>
    <xs:element name="Weibull">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="2" maxOccurs="2" />
      </xs:complexType>
    </xs:element>
    <xs:element name="constant">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="1" maxOccurs="1" />
      </xs:complexType>
    </xs:element>
    <xs:element name="uniform">
      <xs:complexType>
        <xs:group ref="expression" minOccurs="2" maxOccurs="2" />
      </xs:complexType>
    </xs:element>
  </xs:choice>
</xs:group>
```

```
<xs:group name="reference-element">
  <xs:choice>
    <xs:element ref="variable"/>
    <xs:element ref="parameter"/>
    <xs:element ref="observer"/>
    <xs:element ref="event"/>
  </xs:choice>
</xs:group>

<xs:group name="literal">
  <xs:choice>
    <xs:element ref="Boolean"/>
    <xs:element ref="Integer"/>
    <xs:element ref="Real"/>
    <xs:element ref="Symbol"/>
  </xs:choice>
</xs:group>
```

B.2.5 Attributes

```
<xs:element name="attribute">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="expression" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

B.2.6 Types

```
<xs:simpleType name="type-definition">
  <xs:union>
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="Boolean"/>
        <xs:enumeration value="Integer"/>
        <xs:enumeration value="Real"/>
        <xs:enumeration value="Symbol"/>
        <xs:enumeration value="event"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:pattern value="[a-zA-Z_]+[a-zA-Z0-9.:_]*"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
```

B.2.7 Element references

```
<xs:element name="variable">
  <xs:complexType>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="parameter">
  <xs:complexType>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="observer">
  <xs:complexType>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="event">
  <xs:complexType>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

B.2.8 Literal values

```
<xs:element name="Boolean">
  <xs:complexType>
    <xs:attribute name="value" type="boolean-value" use="required"/>
  </xs:complexType>
</xs:element>

<xs:simpleType name="boolean-value">
  <xs:restriction base="xs:string">
    <xs:pattern value="true|false"/>
  </xs:restriction>
</xs:simpleType>

<xs:element name="Integer">
  <xs:complexType>
    <xs:attribute name="value" type="xs:integer" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="Real">
  <xs:complexType>
    <xs:attribute name="value" type="xs:double" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="Symbol">
  <xs:complexType>
    <xs:attribute name="value" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

Appendix C

Revision History

C.1 Differences between AltaRica Data-Flow version 1 and AltaRica Data-Flow version 2

The syntax of AltaRica Data-Flow version 2 is significantly different from the one of the version 1. The semantics of both versions are close, so translating models from the first version to the second one should not be an issue, although it is probably hard to design a fully automatic translator for the whole language.

Here is a list of the main differences between the two versions. In the first version:

- Classes are called “nodes” (and their declaration finishes with the keyword “edon”.
- State variables are declared into a separate clause introduced by the keyword “state”.
- Flow variables are declared into a separate clause introduced by the keyword “flow”.
- Types “Any” and “Symbol” does not exist. Types “Boolean”, “Integer” and “Real” are denoted respectively “bool”, “int” and “float”.
- Instances of other classes are declared into a specific clause introduced by the keyword “sub”.
- State variables are initialized into a special clause introduced by the keyword “init”.
- Flow variables are tagged with attributes “in”, “out” or “private”. This tag which can be easily determined at flattening time is no longer necessary in version 2.
- Probability distributions associated with events are declared in the clause introduced with the keyword “extern”. Their declaration is introduced with the keyword “law”.
- The “expectation” mechanism does not exist. Instead, instantaneous transitions can be grouped into buckets and given a probability. The “expectation” mechanism is more general and much simpler.
- A priority, i.e. a positive integer can be associated with instantaneous events (the lower the number, the higher the priority). This notion has been abandoned in version 2 for at least two reasons: i) it makes the firing mechanism quite complex ii) it is seldom used in practice.
- There is no way to specify the memory policy of an event (policies are all “RESTART”).

- Simple transitions are declared in a separate clause introduced by the keywords “trans”. They are written in the form:
$$G| - e- > P$$
- Synchronizations are declared in a separate clause introduced with the keyword “sync”. They are written in the form:
$$< e : S >$$
where S is a synchronization expression (a Boolean expression over events).
- The assignment is the only instruction. It is written “$v := E$” in actions of transitions and “$v = E$” in assertions.
- Records and functions are available only in some tools. Their syntax is different.
- Parameters are declared in the clause “extern” and have no type.

As an illustration, consider the small system pictured Figure C.1. This system is made of a series of two pairs (main unit, spare unit).

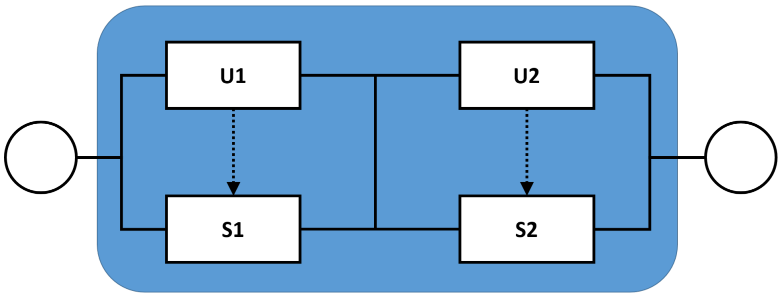


Figure C.1: A system made of a series of two pairs (main unit, spare unit).

The AltaRica Data-Flow version 1 code for this system could be as given Figure C.2. The equivalent code in AltaRica Data-Flow version 2 could be as given Figure C.3. Beside slight syntactic differences, the main issue here is the removal of the extern clause and the integration into the declaration part of the items that were previously declared in the extern clause.

The situation gets trickier with the definition of spare units. The code in AltaRica Data-Flow version 1 for spare units could be as given Figure C.4 while the one in AltaRica Data-Flow version 2 could be as given Figure C.5. The parameter “γ_s” introduced in AltaRica Data-Flow version 1 is no longer needed in version 2 for parameters can be defined by means of expressions. But the main point here stands in the fact that the notion of “bucket” of instantaneous transition has been replaced by the notion of expectation. The latter is much more general and elegant than the former. Moreover, it applies to all types of transitions and not only to instantaneous ones. Therefore, the construction based on buckets and constant laws is advantageously replaced by expectations.

At system level, the two codes are very similar except for the definition of the observer. The different types of observers of version 1 have been unified into a unique concept of typed observer.

```
domain MainUnitState = {WORKING, FAILED};

node MainUnit
    state s:MainUnitState;
    flow inFlow:bool:in; outFlow:bool:out; demandSpare:bool:out;
    event failure, repair;
    trans
        (s=WORKING) |- failure -> s := FAILED;
        (s=FAILED) |- repair -> s := WORKING;
    assert
        outFlow = if (s=WORKING) then inFlow else false;
        demandSpare = (s=FAILED);
    init
        s := working;
    extern
        law <event failure> = exponential(lambda);
        law <event repair> = exponential(mu);
        parameter lambda = 1.0e-4;
        parameter mu = 1.0e-2;
edon
```

Figure C.2: AltaRica Data-Flow (version 1) code for the main unit.

```
domain MainUnitState {WORKING, FAILED}

class MainUnit
    MainUnitState s (init = WORKING);
    Boolean inFlow, outFlow (reset = false);
    Boolean demandSpare (reset = false);
    event failure (delay = exponential(lambda));
    event repair (delay = exponential(mu));
    parameter Real lambda = 1.0e-4;
    parameter Real mu = 1.0e-2;
    transition
        failure: s == WORKING -> s := FAILED;
        repair: s == FAILED -> s := WORKING;
    assertion
        outFlow := if s == WORKING then inFlow else false;
        demandSpare := s == FAILED;
end
```

Figure C.3: AltaRica Data-Flow (version 2) code for the main unit.

```
domain SpareUnitState = {IDLE, WORKING, FAILED};

node spare
    state s:SpareUnitState ;
    flow inFlow:bool:in; outFlow:bool:out; demanded:bool:in;
    event failureOnDemand, turnOn, turnOff, failure, repair;
    trans
        ((s=IDLE) and demanded) |- turnOn -> s := WORKING;
        ((s=IDLE) and demanded) |- failureOnDemand -> s := FAILED;
        ((s=WORKING) and (not demanded)) |- turnOff -> s := IDLE;
        (s=WORKING) |- failure -> s := FAILED;
        (s=FAILED) |- repair -> s := IDLE;
    assert
        outFlow = if (s=WORKING) then inFlow else false;
    init
        s := IDLE;
    extern
        law <event failureOnDemand> = constant(gamma);
        law <event turnOn> = constant(gamma_s);
        law <event failure> = exponential(lambda);
        law <event repair> = exponential(mu);
        parameter gamma = 0.01;
        parameter gamma_s = 0.99;
        parameter lambda = 1.0e-4;
        parameter mu = 1.0e-2;
        bucket {<event failureOnDemand>, <event start>} = call;
edon

node System
    sub U1:MainUnit; U2:MainUnit; S1:SpareUnit; S2:SpareUnit;
    assert
        U1.inFlow = true,
        S1.inFlow = true,
        S1.demanded = U1.demandSpare,
        U2.inFlow = (U1.outFlow or S1.outFlow),
        S2.inFlow = (U1.outFlow or S1.outFlow),
        S2.demanded = U2.demandSpare;
    extern
        predicate failed = <term (not (U2.o or S2.o))>;
edon
```

Figure C.4: AltaRica Data-Flow (version 1) code for the spare unit and the system.

```
domain SpareUnitState {IDLE, WORKING, FAILED}

class SpareUnit
    SpareUnitState s (init = IDLE);
    Boolean inFlow, outFlow (reset = false);
    Boolean demanded (reset = false);
    parameter Real lambda = 1.0e-4;
    parameter Real mu = 1.0e-2;
    parameter Real gamma = 0.01;
    event failure (delay = exponential(lambda));
    event repair (delay = exponential(mu));
    event turnOn (delay = Dirac(0), expectation = 1-gamma);
    event failureOnDemand (delay = Dirac(0), expectation = gamma);
    event turnOff (delay = Dirac(0));
    transition
        turnOn: s == IDLE and demanded -> s := WORKING;
        failureOnDemand: s == IDLE and demanded -> s := FAILED;
        turnOff: s == WORKING and not demanded -> s := IDLE;
        failure: s == WORKING -> s := FAILED;
        repair: s == FAILED -> s := WORKING;
    assertion
        outFlow := if s == WORKING then inFlow else false;
end

class System
    MainUnit U1, U2;
    SpareUnit S1, S2;
    observer Boolean failed = not (U2.o or S2.o);
    assertion
        U1.inFlow := true;
        S1.inFlow := true;
        S1.demanded := U1.demandSpare;
        U2.inFlow := (U1.outFlow or S1.outFlow);
        S2.inFlow := (U1.outFlow or S1.outFlow);
        S2.demanded := U2.demandSpare;
end
```

Figure C.5: AltaRica Data-Flow (version 2) code for the spare unit and the system.

C.2 Differences between AltaRica 3.0 and AltaRica Data-Flow version 2 (normalized)

The main difference between these two versions AltaRica 3.0 and AltaRica Data-Flow (normalized) are on two ways. First AltaRica 3.0 introduces new constructs to structure models (presented into chapter 10). Second the underlying mathematical model (Guarded Transitions Systems presented into chapter 3) can handle acausal component and systems with instant loops.

Index