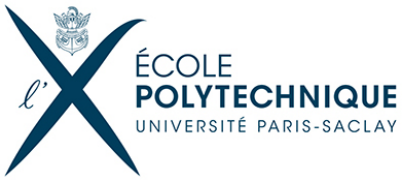
AltaRica 3.0: a Model-Based approach for Safety Analyses

Tatiana Prosvirnova



Thèse présentée pour obtenir le grade de

DOCTEUR DE L'ECOLE POLYTECHNIQUE

Spécialité : Informatique

Tatiana PROSVIRNOVA

AltaRica 3.0 : une approche orientée modèles pour la Sûreté de Fonctionnement

---

AltaRica 3.0: a Model-Based approach for Safety Analyses

Soutenue publiquement le 21 Novembre 2014 à l'Ecole Polytechnique
devant le jury composé de :

Président du jury	:	**Leïla**	**KLOUL**	Université de Versailles St-Quentin-en-Yvelines
Rapporteurs	:	**Mohamed**	**KAANICHE**	Laas-CNRS, Toulouse
		Olivier	**ROUX**	IRCCyN, Ecole Centrale de Nantes
Directeur de thèse	:	**Antoine**	**RAUZY**	Ecole Polytechnique, Palaiseau
Examinateurs	:	**Michel**	**BATTEUX**	IRT SystemX, Palaiseau
		Frank	**ORTMEIER**	Otto-von-Guericke University of Magdeburg
		Christel	**SEGUIN**	ONERA, Toulouse

Tatiana PROSVIRNOVA

AltaRica 3.0 : une approche orientée modèles pour la Sûreté de Fonctionnement

---

AltaRica 3.0: a Model-Based approach for Safety Analyses

2014

Laboratoire d'Informatique (LIX)
Ecole Polytechnique
France

A mes parents et mon mari...

Моим родителям и мужу посвящается...

Remerciements

2007 – Découverte du langage AltaRica pendant le stage à Thales Research & Technology.
2008 – Rencontre avec Antoine Rauzy, l'un des créateurs du langage AltaRica.
2011 – Début de la thèse sous la direction d'Antoine Rauzy sur la nouvelle version du langage AltaRica.

A l'issue de ma scolarité à l'Ecole Polytechnique je n'avais pas forcément envie de poursuivre une thèse. Mais j'ai eu l'opportunité de faire partie d'un projet de recherche innovant ayant de réelles applications industrielles. Je tiens à remercier mon directeur de thèse, Antoine Rauzy, pour cette expérience unique que j'ai vécue durant ces trois années. Un grand merci pour m'avoir permis de travailler dans un contexte scientifique de grande qualité, d'avoir cru en moi d'abord en m'embauchant chez Dassault Systèmes et ensuite en m'offrant la possibilité de faire une thèse à l'Ecole Polytechnique.

Je voudrais également remercier Leïla Kloul avec qui j'ai collaboré pendant ma thèse. Tu m'as permis de découvrir de nouveaux domaines de recherche, ainsi que l'enseignement à l'Université de Versailles.

Un grand merci à Jean-Marc Roussel pour sa pédagogie. Tes conseils très pertinents m'ont été très utiles pour ma soutenance de thèse.

Un grand merci aussi à Michel Batteux. Ton aide pour ma soutenance de thèse et pour mon manuscrit, ainsi que ton soutien tout au long de ces trois années, m'ont été très précieux.

Je tiens également à remercier mes rapporteurs Mohamed Kaâniche et Olivier Roux ainsi que tous les autres membres de mon jury Leïla Kloul, Michel Batteux, Frank Ortmeier et Christel Seguin. Vous m'avez tous fait un grand honneur en acceptant d'être présents à ma soutenance et en prenant le temps de lire attentivement mon manuscrit. Vos questions et remarques m'ont été très précieux pour améliorer mon travail et approfondir encore mes connaissances dans le domaine.

Ma reconnaissance va aussi à tous les membres de l'équipe AltaRica 3.0, que j'ai pu côtoyer depuis septembre 2011. Chacun d'entre vous a contribué à sa manière au résultat que vous voyez aujourd'hui. Merci à nos doctorants Pierre-Antoine Brameret, Thomas Friedlhuber, Abraham Cherfi, Melissa Issad et Huixing Meng pour leur aide, leurs conseils et les moments que nous avons partagés ensemble pendant les conférences, les séminaires et les petites fêtes. Merci à nos stagiaires Renaud Lancelot, Kseniïa Isaeva, Hala Mortada et Nawaal Mamadou pour leur travail, leur bonne humeur et les moments que j'ai partagés avec eux.

Mais le labo ne se résume pas qu'au groupe AltaRica 3.0. Je tiens à remercier aussi notre secrétaire Evelyne Rayssac et notre informaticien James Regis sans qui le labo ne fonctionnerait pas aussi bien.

A ce sujet, merci à l'Ecole Doctorale (EDX) pour son soutien financier pendant ces trois ans à travers l'allocation internationale de thèse Gaspard Monge. Plus particulièrement, merci à Fabrice et Audrey pour leur disponibilité et leur écoute.

Je souhaite enfin exprimer toute ma gratitude envers le professeur Frank Ortmeier et ses doctorants Michael Lipaczewski et Simon Struck qui m'ont accueilli et fait découvrir leur culture lors de mes deux séjours en Allemagne en 2012 et en 2013.

Je voudrais remercier mon maître de stage Stéphane Mallat qui m'a fait découvrir les domaines de l'ingénierie dirigée par des modèles et de la sûreté de fonctionnement.

Merci à Marc Bouissou pour le partage de son expertise en sûreté de fonctionnement.

Je voudrais également remercier tous mes anciens collègues de Dassault Systèmes, en particulier

Benoît Perrot pour m'avoir donné goût à la programmation.

Je tiens à remercier tous mes amis qui ont toujours été à mes côtés pour me soutenir. Merci à Cyril et Alexandra qui ont commencé leurs thèses en même temps que moi mais dans un domaine complétement différent, pour tous les moments que nous avons vécus ensemble durant ces trois années. Merci à Nicolas pour ses conseils et ses idées. Je remercie également mes amis de Master d'Ingénierie des Systèmes Industriels Complexes Marcel, Jujhar et Mounir. Un grand merci à mes amis Sunanda et Juan qui ont soutenu leurs thèses quelques mois avant moi. Merci à Gaela, Cécile et Hélène pour les moments que nous avons partagés au badminton. Merci à Erwana pour son soutien et son amitié dès le début de mon arrivée en France. Merci à Charlotte que je connais depuis très longtemps et que j'ai retrouvée à Paris durant ma thèse.

Je voudrais remercier ma belle famille et mon mari Cyril qui m'a toujours soutenu et aidé tout au long de cette aventure. Tu m'as montré la voie, m'a aidé avec tes conseils et a toujours été pour moi un exemple à suivre. Merci à ce cours d'anglais du 10 septembre 2007 qui nous a permis de se rencontrer. Merci à notre bébé qui me donnait des petits coups de pied le jour de ma soutenance, le 21 novembre 2014.

Я также хотела бы поблагодарить моих друзей из Бауманки Иру,Таню, Женю, Колю, Сашу и Свету за их советы и поддержку.

Огромное спасибо моим родителям, за то, что всегда верили в меня, помогали и поддерживали во всех моих начинаниях.

Le 21 février 2015, à Toulouse.
Tatiana.

Introduction

Safety and Reliability of systems is of great importance for environmental, social and economic reasons. Whether it be for nuclear engineering or for the design of new means of transport (like the Falcon or the Shinkansen), system designers have to perform Safety and Reliability Analyses from the earliest phases of the project. These analyses are codified by the regulation authorities through safety standards such as IEC 61508, ISO 26262, ARP4754 or ARP4761.

Reliability engineers have developed various methods of risk analysis which are now well-mastered: Failure Modes and Effects Analyses (FMEA), Fault Tree Analyses [6], Event Tree Analyses [107], etc. Efficient algorithms and powerful assessment tools are available for them. However, these formalisms have a major drawback. They rely on too low level modeling formalisms. As a consequence, there is always a gap between the specification of the system under study and the associated safety models. This gap makes safety models hard to share amongst the stakeholders and to maintain throughout the life cycle of systems. Even a minor change in the specification may require a complete review of the safety model which is both resource consuming and error prone.

Nowadays, Model-Based Safety Assessment (MBSA) – the Reliability Engineering declension of Model-Based System Engineering – focuses more and more attention in the world. The idea is to write models in high level description formalisms so as to keep them close to the functional and physical architecture of the system under study. High level models can be processed directly (typically by stochastic simulation) or automatically compiled into a lower level formalism (e.g. a Fault Tree).

AltaRica is such a high level modeling language dedicated to Safety Analysis. The first version of the language has been created in the Computer Science Laboratory of the University of Bordeaux (LaBRI) at the end of nineties [80, 7]. This first version made it possible to set-up the basic concepts but was too resource consuming for industrial scale models. Therefore, a second version, AltaRica Data-Flow, has been created at Institute of Mathematics of Luminy (IML, Marseilles) a few years later [88, 14]. AltaRica Data-Flow is now in the core of several industrial Integrated Modeling and Simulation Environments: Cecilia OCAS (Dassault Aviation), Simfia (EADS Apsys) and Safety Designer (Dassault Systemes). In addition, a great number of tools have been developed to assess AltaRica Data-Flow models, such as Fault Tree compilers, compilers to Markov chains, generators of critical sequences of events, stochastic simulators and model-checkers. AltaRica Data-Flow Integrated Modeling and Simulation Environments are widely used across various industries. Many successful industrial applications have been reported in the literature [10, 97, 52, 85, 4]. AltaRica Data-Flow has now reached an industrial maturity.

AltaRica is an event-based modeling language. Deterministic or stochastic delays can be associated with events. The behavior of components is described by means of state machines. The state of a component is represented by variables (so called state variables) and their values. The changes of state are possible when, and only when, an event occurs. They are described by the transitions. Flow variables are used to model information circulating between components. They are updated by the assertion, which is executed after each transition firing. Components can be assembled into hierarchies, their inputs and outputs can be connected and their transitions can be synchronized.

The aim of the AltaRica 3.0 project [82], conducted at the Computer Science Laboratory of Ecole Polytechnique (LIX), is to develop a modeling, simulation and assessment platform to perform Safety

Analyses with AltaRica 3.0 modeling language. The new version of the AltaRica language, the so-called AltaRica 3.0, is in the core of this project. AltaRica 3.0 improves AltaRica Data-Flow into two directions. First, its underlying mathematical model is based on Guarded Transition Systems (GTS). Second, the language provides new constructs to structure models.

The project aims to develop the following assessment tools (see Figure 1):

- The compiler from AltaRica 3.0 to Guarded Transition Systems;
- The stepwise simulator for Guarded Transition Systems;
- The graphical simulator of AltaRica 3.0 models;
- The compiler from Guarded Transition Systems to Fault Trees;
- The Fault Tree assessment tool XFTA;
- The Sequence Generator for Guarded Transition Systems;
- The compiler from Guarded Transition Systems to Markov chains;
- XMRK, a tool to assess multi-phase Markov chains with rewards;
- The Stochastic Simulator for Guarded Transition Systems;
- The Model-checker for Guarded Transition Systems;
- The Reliability allocation module for Guarded Transition Systems.

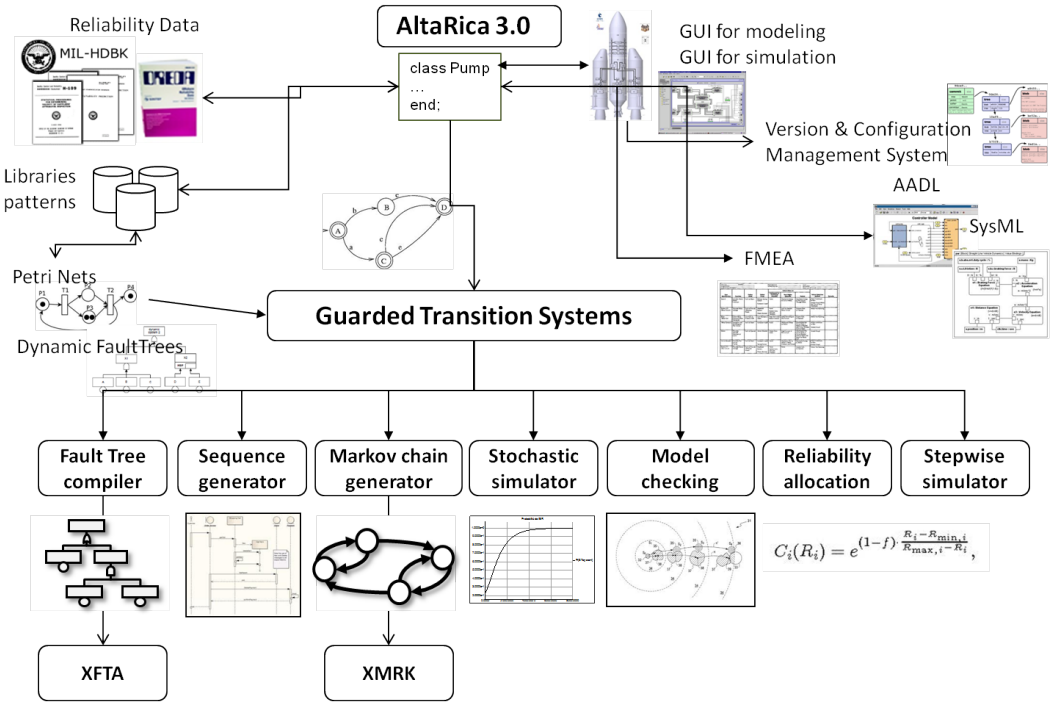


Figure 1: The AltaRica 3.0 project

These tools enable the user to perform virtual experiments on systems, to perform end-to-end risk assessment with AltaRica 3.0 and also to do cross check calculations. Thanks to these tools

AltaRica models can be used to perform Preliminary System Safety Analysis (PSSA) and System Safety Analysis (SSA).

In other words, with AltaRica 3.0 models, it will be possible:

- To perform Fault Tree Analysis (FTA) for static and some kinds of dynamic models;
- To calculate different probabilistic indicators for dynamic models using Markov chain analysis;
- To perform stochastic simulation of dynamic models;
- To verify system and model properties using model-checking techniques;
- To graphically simulate the model in order to validate it;
- To perform reliability and availability allocation for different components given the overall objective.

Within the AltaRica 3.0 project, it is also planned to develop bridges with other tools, especially to work on the integration of system architecture with Safety Analyses through the development of methods and tools to synchronize models of both disciplines.

The contribution of this PhD thesis to the AltaRica 3.0 project is as follows:

- Chapter 2 presents Guarded Transition Systems (GTS), the new underlying mathematical model of AltaRica 3.0. In addition to the ability to handle looped systems thanks to the introduction of a fixpoint mechanism to calculate assertions, as proposed in [90], GTS make it possible to define acausal components (components for which the input and output flows are decided at run time). Different algorithms to calculate assertions and to optimize them are discussed. They have been implemented in the stepwise simulator of GTS. Experiments have been performed, e.g. modeling of network systems, as reported in [71].
- Chapter 3 introduces the structural constructs of AltaRica 3.0. These new structural constructs come from object-oriented and prototype-oriented modeling languages. From our point of view, they make it possible to match better with cognitive processes of engineers. They are assembled into System Structure Modeling Language (S2ML). A new algorithm to flatten hierarchical models, i.e to collapse a hierarchy of nested blocks and instances of classes into a single block, is proposed. It has been implemented in the compiler of AltaRica 3.0 models to Guarded Transition Systems.
- Chapter 4 describes the principle of compilation of Guarded Transition Systems to Fault Trees and critical sequences of events. The compilation algorithm is presented in details. It has been implemented in the compiler of Guarded Transition Systems to Fault Trees. Some experiments are reported.
- Chapter 5 presents the overall architecture of the modeling, simulation and assessment platform developed within the AltaRica 3.0 project. It pays a particular attention to the tools developed as a part of this PhD thesis.
- The series of appendices regroups additional works on AltaRica 3.0 done during this PhD thesis.

Outline of the thesis

To summarize, this thesis is organized in 5 chapters:

- Chapter 1 gives an overview of the main concepts and of the state-of-the-art modeling languages and tools dedicated to Safety Analyses.

- In chapter 2, we introduce Guarded Transition Systems (GTS), the underlying mathematical formalism of AltaRica 3.0.
- In chapter 3, we describe structural constructs of AltaRica 3.0, assembled into System Structure Modeling Language (S2ML).
- In chapter 4, we present the algorithm of compilation of GTS into Fault Trees and critical sequences of events.
- Chapter 5 describes the architecture of the AltaRica 3.0 Modeling, Simulation and Assessment platform.

Finally, this manuscript ends with a series of appendices, which regroup additional works on AltaRica 3.0 done during this PhD thesis:

- Appendix A is dedicated to the modeling of systems with mobile components, e.g. production chains or mobile networks. It presents a comparison between AltaRica and PEPA (Performance Evaluation Process Algebra) nets – a modeling formalism for mobility modeling.
- In appendix B, we compare AltaRica with SAML (Safety Analysis Modeling Language).
- Appendix C gives an overview of GraphXica – a high level modeling language for graphical representation and animation of models. It also describes the graphical simulation of AltaRica 3.0 models.
- In appendix D, we present some modeling patterns to represent common cause failures, cold redundancies, system reconfiguration and shared resources on the example of an electrical system.

Contents

List of Figures

List of Tables

Chapter 1

Model-Based Safety Assessment

The goal of this chapter is to give an overview of the concepts, modeling formalisms and assessment tools related to the Safety Analysis of critical systems. First, we introduce some basic notions of Safety and Reliability studies. Then, we present the traditional approaches to perform Safety Analyses and compare some classical modeling formalisms: the Fault Trees, the Reliability Block Diagrams, the Markov chains and the Generalized Stochastic Petri nets. The third section is dedicated to Model-Based Safety Assessment. We present the advantages of this approach and give an overview of the existing high level modeling languages dedicated to Safety Analyses. Finally, in the last section, we introduce AltaRica – a high level modeling language dedicated to Safety Analyses, which is the core of this PhD thesis.

1.1 Safety and Reliability studies

Safety and Reliability of systems is of great importance for environmental, social and economic reasons. System designers have to perform Safety and Reliability Analyses from the earliest phases of their projects. These analyses are codified by regulation authorities through safety standards such as:

- IEC 61508 (Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems), intended to be a basic functional safety standard applicable to all kinds of industry;
- IEC 61839 (Probabilistic Risk Assessment (PRA));
- IEC 60812 (Failure Mode and Effects Analysis (FMEA));
- IEC 61025 (Fault Tree Analysis (FTA));
- ISO 26262, an adaptation of IEC 61508 for Automotive Electric/Electronic Systems;
- IEC 62279, a specific interpretation of IEC 61508 for railway applications;
- DO 178B (Software considerations in airborne systems and equipment certification), etc.

Risk has a bi-dimensional nature. Any threat against the system under study must be analyzed along two criteria: its frequency and its severity. That is why, most of the Safety and Reliability Analyses rely on a probabilistic approach: the goal is to determine the most frequent and severe failure scenarios. If the risk is considered as too high and thus unacceptable, then safety mechanisms can be used to minimize its frequency, its severity or both.

The goal of Safety and Reliability studies is to ensure that the system under study satisfies the safety requirements. Safety requirements can be of different nature:

- Qualitative, e.g. "no single failure should lead the system to its failure state", and

- Quantitative, e.g. "the probability of system failure should be less than 10^{-9}".

In other words, Safety and Reliability studies aim to determine different failure scenarios leading the system from the nominal state to its failure state and to assess different indicators, e.g. the probability of system failure.

Safety Analyses are divided into two groups:

- Qualitative analysis, which goal is to determine different failure scenarios leading the system from its nominal state to its failure state;
- Quantitative analysis, which aim is to assess different probabilistic indicators, such as the probability of system failure.

1.1.1 Probabilistic indicators

Different probabilistic indicators have been introduced in Safety and Reliability studies [8]:

- The **Reliability** of a system or a component is its ability to function under stated conditions for a specified period of time. Let T be a random variable representing the time of well functioning of a component or of a system. The reliability is denoted by $R(t)$ and is calculated as follows:

  $$R(t) = P[T > t]$$

  This indicator is of paramount importance for non-repairable systems. Sometimes the **Unreliability** is considered instead of the reliability. It is defined as follows: $\overline{R}(t) = 1 - R(t)$.

- The **Availability** of a system or a component is its ability to function under stated conditions at a specified time t. Let X be a random variable which is equal to 1 at time t if the system is operational at time t and it is equal to 0 otherwise. Then the availability, denoted by $A(t)$, is calculated as follows:

  $$A(t) = P[X(t) = 1]$$

  The availability is important for systems with repairable components. In some situations, the **Unavailability** is used instead of the availability. It is calculated as follows: $\overline{A}(t) = 1 - A(t)$.

- The **Maintainability** of a system or a component is its ability to be repaired before a given time t. Let T_M be a random variable, representing the time needed to repair a component or a system. Considering that a system or a component is failed at time $t = 0$, the maintainability, denoted $M(t)$, is calculated as follows:

  $$M(t) = P[T_M < t]$$

One often considers the average values of the indicators:

- Mean Time To Failure (MTTF), the average value of the **Reliability**;
- Mean Time To Repair (MTTR), the average value of the **Maintainability**.

Probability distributions are used to model failures and repairs of individual components. The most commonly used distribution is the **exponential** distribution

$$F(t) = 1 - e^{-\lambda t}, t \geq 0,$$

where λ is called a failure rate. The same distribution with a repair rate μ instead of λ is often used to represent repairs.

1.1.2 Redundancies

In order to reduce the risk of system failure, redundancies can be introduced in the systems: as a result, the system under study is composed of a "main component" and one or more "spare components". The spare components are in "standby" state (also called dormant state), if the main component is "operational" (is in "active" state). When the main component fails, it is replaced by a spare component which then becomes active. A spare component may fail in both the dormant and the active states. However, in standby mode, the failure rate is reduced by a factor α, called the dormancy factor, which takes values in $[0; 1]$. According to the value of α, a spare component may be:

- In **Cold redundancy**: that means that the spare component cannot fail in standby mode and it corresponds to the case when $\alpha = 0$;
- In **Hot redundancy**: the spare component is working at the same time as the main component ($\alpha = 1$);
- In **Warm redundancy**: if $0 < \alpha < 1$, that means that the spare component may, however, fail in standby mode.

Spare components may be shared between several main components. If a spare component has already replaced a main component, it cannot replace another component.

1.1.3 Safety Assessment

In order to perform Safety Analyses of a given system, Safety Analysts proceed in three steps as illustrated in Figure 1.1.

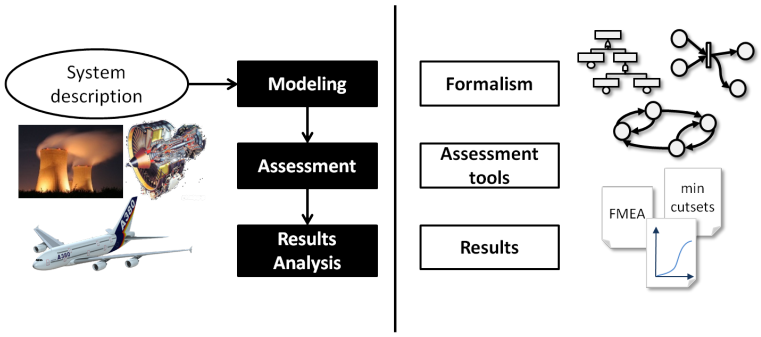


Figure 1.1: Safety Assessment

The Modeling: First of all, given a system description, the safety analyst creates an appropriate safety model of the system under study. The choice of the formalism depends on the system to model and on the requirements to verify. In practice, this choice depends on the available assessment tools. In the following sections we will see different formalisms used for Safety Analyses.

The Assessment: The second step consists in assessing the created model in order to calculate failure scenarios and different probabilistic indicators.

The Results analysis: Finally, the engineer analyzes the obtained results and produces a report describing if the system satisfies the given safety requirements.

1.2 Classical approach for Safety Analysis

Formalisms, traditionally used for Safety and Reliability studies, can be classified in two categories:

- Boolean formalisms, such as Fault Trees [6], Event Trees [107], Reliability Block Diagrams [6].
- States/Transitions formalisms such as Markov chains [101] and Stochastic Petri Nets [63].

Boolean formalisms are the most popular formalisms used for Safety and Reliability studies. Efficient assessment algorithms have been developed for this category of models. They give good approximations of system behaviour. However, these models are not able to represent dependencies between failures and thus cannot capture phenomena such as system reconfigurations or shared resources.

States/Transitions formalisms have an improved expressive power: they allow to capture the order of appearance of events and to represent different types of dependencies. They are a good trade-off between the expressive power and the efficiency of assessment algorithms. Many states/transitions formalisms have been proposed in the literature. Special assessment algorithms have been developed for each of these formalisms.

In this section we present some classical modeling formalisms for Safety Analysis and discuss their advantages and drawbacks. Section 1.2.1 is dedicated to Boolean formalisms for Safety Analysis. Fault Trees and Reliability Block Diagrams are presented. Section 1.2.2 introduces States/Transitions formalisms used for Safety studies. Section 1.2.3 presents some extensions of classical formalisms for Safety Analysis.

1.2.1 Boolean Formalisms

Boolean formalisms are most commonly used in Safety and Reliability studies of industrial systems. They are simple, cover a large spectrum of modeling problems. In addition, very efficient assessment algorithms are available for them.

This category of formalisms mainly includes:

- Fault Trees (FT) [6],
- Event Trees (ET) and
- Reliability Block Diagrams (RBD) [6].

In this section we present Fault Trees and Reliability Block Diagrams. The interested reader can refer to [107] for a detailed description of Event Trees.

Fault Trees (FT)

Fault Tree Analysis (FTA) [6] is the most commonly used method for Safety Analysis of industrial systems.

A Fault Tree is a graphical representation of the relationships between the failure events of the modeled system. The root of the Fault Tree, called the "*top event*", represents the global system failure (the undesirable event). The leaves of the Fault Tree, called "*basic events*", represent the failures of the individual components. Probability distributions can be associated with basic events. Basic events are connected to the top event by means of intermediate events and gates. A "*gate*" is a logical operator: it has several Boolean inputs and one Boolean output. The most commonly used gates are: OR, AND, K-out-of-N (see [6] for a more detailed description). "*Intermediate events*" are used to name the outputs of gates.

Example 1.1 (A fault-tolerant multiprocessor system)**.** Consider a fault-tolerant multiprocessor system depicted Figure 1.2. The system is inspired from [63]. It is made of two redundant processing subsystems S1 and S2, a shared memory bank M3 and a bus N. Each subsystem consists of a processor, a local memory bank and two redundant disks. Both processors have access to a shared memory bank M3, through a bus N. We assume that each component may fail in operation and its probability of failure is exponentially distributed with a failure rate $\lambda = 0.001$.

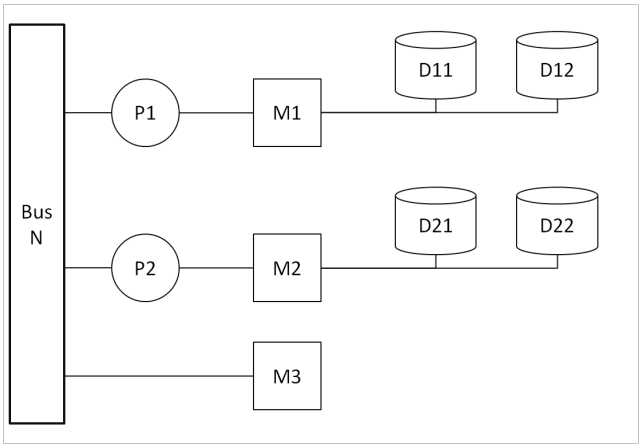


Figure 1.2: A fault-tolerant multiprocessor system

The goal is to determine failure scenarios leading the system from the nominal state (when all components are operational) to a failure state (when the system cannot process data) and to assess the probability of the system failure.

The Fault Tree representing this system is depicted in Figure 1.3. Basic events are represented by rectangles with circles below and are marked in grey. They model failures of individual components: memories, processors and disks. The top event, *"System failed"*, occurs when the bus N is failed or subsystems S1 and S2 are both failed. So it is connected to the basic event *"N failed"* and to the intermediate event *"S1 & S2 failed"* via an OR gate. The intermediate event *"S1 & S2 failed"* is then further broken down. It is connected via an AND to another two intermediate events *"S1 failed"* and *"S2 failed"*. Each of these intermediate events is then connected to basic events via OR and AND gates.

Note that a Fault Tree is represented by a Directed Acyclic Graph (DAG), thus the same leaf can be shared between several branches of the tree (see, for example, the basic event *"M3 failed"* Figure 1.3).

Reliability Block Diagrams (RBD)

In Reliability Block Diagrams, the logic diagram is arranged to indicate the combinations of properly working components keeping the system operational. A Reliability Block Diagram is a set of blocks connected together in parallel or in series. Blocks connected together can be assembled into hierarchies of blocks. Blocks represent system components or subsystems. Each block can be only in two states: working or failed. Only two types of connections between blocks are considered: in parallel or in series. They can be combined.

When modeling a system, it should be abstracted into a hierarchy of blocks connected in parallel or in series.

Example 1.2 (A fault-tolerant multiprocessor system)**.** The RBD representing the fault-tolerant multiprocessor system, described earlier, is given Figure 1.4. The block N represents the bus N and

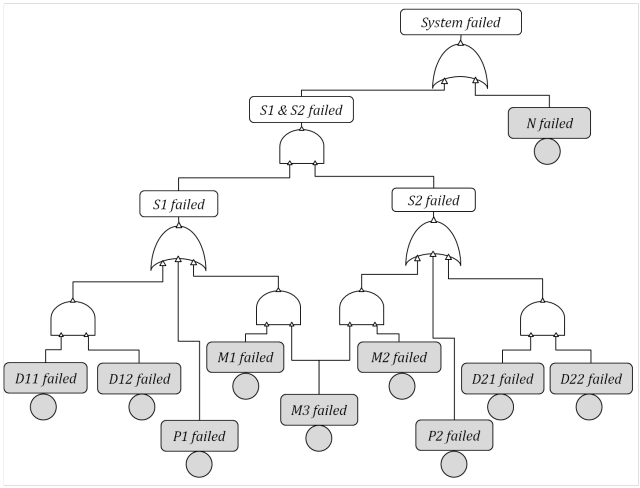


Figure 1.3: Fault Tree of the fault-tolerant multiprocessor system

the hierarchical blocks *S1* and *S2* represent the subsystems S1 and S2, respectively. The blocks *S1* and *S2* are connected in parallel. They are connected in series with the block *N*. Each hierarchical block *Si*, i=1,2, is then broken down into blocks connected together in parallel or in series. Blocks representing disks (*Di1* and *Di2*, i=1,2) are connected in parallel, thus they are redundant. Blocks representing memories are also connected in parallel. Block *Pi*, i=1,2, parallel blocks *Di1* and *Di2*, and parallel blocks *Mi* and *M3* are connected in series.

Note that the block *M3* is the same in the blocks *S1* and *S2*.

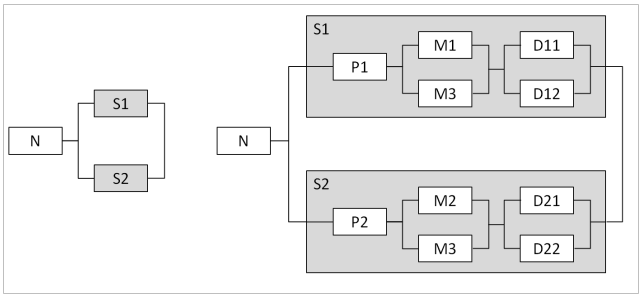


Figure 1.4: RBD for the fault-tolerant multiprocessor system

Analysis and assessment tools

Reliability Block diagrams can be easily transformed into Fault Trees [99]. A Fault Tree encodes a set of Boolean equations. Both qualitative and quantitative analysis can be performed on a Fault Tree. Qualitative analysis consists of calculation of minimal cutsets. A minimal cutset (MCS) is the smallest combination of basic events that leads to the top event (see [87] for a mathematical definition of a minimal cutset).

Quantitative analysis mainly consists of assessing the probability of the top event and the probabilities of the intermediate events, knowing the probability distributions of basic events. Fault Trees can be used to calculate Safety Integrity Levels (SIL) [31] and importance factors [32].

Very efficient algorithms have been developed to assess Fault Trees with up to several thousand Basic Events (see e.g. [93, 91]). Some of them are based on symbolic representations of FT as binary decision diagrams (BDD) as proposed in [86].

A lot of commercial RAMS (Reliability, Availability, Maintainability, Safety) workbenches are available: Aralia Fault Tree Analyzer (Dassault Systemes), FaultTree+ (Isograph), BlockSim (ReliaSoft Corporation), Item Toolkit (ITEM Software), CAFTA (Electric Power Research Institute), etc. In general, these workbenches include a graphical user interface and assessment tools to calculate minimal cutsets and probabilities of events.

In 2008 an Open-PSA initiative was launched: its objective is to create a model exchange format for PSA models [33, 51]. The main idea is to be able to exchange models (Fault Trees, Event Trees, etc.) between different assessment tools in order to perform cross check verifications.

Advantages and drawbacks

Boolean formalisms present many advantages. First of all, they are event-based. However, only failure events can be considered. Other events (e.g. repairs, reconfigurations) cannot be represented explicitly.

Second, they are naturally hierarchical. They make it possible to structure models into hierarchies and to represent break-down structures, as previously seen in Figures 1.3 and 1.4.

Third, they make it possible to represent remote interactions between components, what is especially clear in case of Reliability Block Diagrams.

Finally, they have convenient graphical representations which is important for industrial scale models. Very efficient algorithms and tools are available. Indeed, Boolean formalisms are a good trade-off between the expressive power of the formalism and the efficiency of the assessment algorithms.

However, Boolean formalisms put very strong constraints on events (failures) to be considered. All events are assumed to be statistically independent. Among other consequences, it is not possible to take into account the order in which events occur and events can occur any time, no matter the current state of the system.

In the system, described in Example 1.1 and depicted in Figure 1.2, consider that the memory bank M3 is a spare unit shared by the subsystems S1 and S2. When the memory banks M1 and M2 are operational, M3 is not used. When M1 is failed, it is replaced by M3, and M3 cannot be used to replace M2 anymore. In that case the order of occurrence of events is important. Indeed, the event *"M3 failed"* cannot occur before the failures of the memory banks M1 and M2. The events can no longer be considered as independent.

This type of systems is called *dynamic*, converse to *static* systems, i.e. systems for which all events are considered to be independent. Boolean models give approximated results for dynamic systems. To get more precise results, one needs to use more expressive modeling formalisms (e.g. States/Transitions formalisms described in Section 1.2.2).

Also, from a system engineering point of view, Fault Trees and Reliability Block Diagrams are too low level modeling formalisms. Consequently, there is always a gap between the specifications of the systems under study and the associated safety models. This gap makes safety models hard to design, to share amongst stakeholders and to maintain throughout the life cycle of systems. Even a minor change in the specifications may require a complete review of the safety models, which is time consuming, costly and error prone. It is also difficult to ensure the traceability between system specifications and safety models.

The advantages and drawbacks of Boolean formalisms are summarized in Table 1.1.

1.2.2 States/Transitions Formalisms

States/transitions formalisms make it possible to capture dependencies amongst components, such as cold redundancies, resources sharing and sequences of actions. They can handle dynamic models. This greater expressive power comes indeed with a price in terms of practical calculability. They should be used when approximations made with Boolean formalisms are not suitable for the problem under study. This category basically includes:

Formalism	Advantages	Drawbacks
Boolean formalisms	Event-based Hierarchical Remote interactions Graphical representation Efficient assessment algorithms	Low expressive power Low level formalism (far from system specifications) Hard to design and to maintain

Table 1.1: Boolean formalisms

- Markov Chains (MC), and
- Generalized Stochastic Petri Nets (GSPN).

Markov Chains (MC)

Markov chains [101] used for Safety Analyses are probabilistic finite state machines. They have a convenient graphical representation:

- System states are represented by circles;
- Transitions between states are represented by arrows labeled by the probabilities. These probabilities typically correspond to the failure rate λ or to the repair rate μ of system components.

Some states are considered as operational for the system under study (some components may be failed in these states), others are considered as failure states.

To be represented by a Markov chain, the system should verify Markov assumption: "System evolution depends only on the current state of the system". In other words, the process is without memory. The assumption is very strong. But it is verified if the delays associated with components failures and repairs are exponentially distributed.

Figure 1.5 illustrates a Markov chain representing a repairable component.

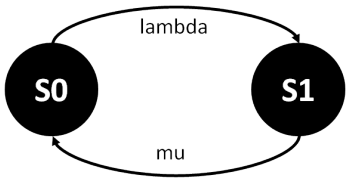


Figure 1.5: Markov chain representing a repairable component

Generalized Stochastic Petri Nets (GSPN)

When using Generalized Stochastic Petri Nets [63] to perform Safety Analyses of systems, places can be interpreted as system states and transitions are often associated with events. A delay possibly stochastic is associated with each transition. Transitions may be immediate or timed. When there are several immediate transitions fireable at a time, the choice is done according to the probability associated with each fireable transition. To be valid the sum of probabilities of all fireable immediate transition should be equal to 1.

Figure 1.6 shows a Petri net representing a repairable component. There two places representing the working state (W) and the failure state (F). Events "failure" and "repair" are associated with

transitions. These transitions are timed. It is possible to define their probability distributions. For example, it can be exponential distributions with a failure rate λ and a repair rate μ. In general, GSPN are assessed by stochastic simulation.

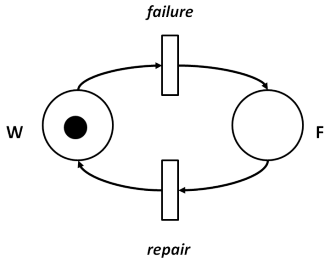


Figure 1.6: Petri Net representing a repairable component

Analysis and assessment tools

Basically, there are three methods to assess dynamic models:

- If the model verifies Markov assumption, then it can be assessed by Markov analysis, in order to calculate reliability indicators.
- When the model does not verify Markov assumption or is too big to be assessed by Markov analysis, stochastic simulation can be used to calculate reliability indicators.
- It is possible to generate critical sequences of events. The limit can be the length, also called the order, of the sequence or its probability.

Each of these methods has its advantages and drawbacks.

Markov analysis The corresponding Markov chain is transformed into a system of differential equations of first order with constant coefficients [101]. From this system of differential equations, it is possible to calculate:

- the three most important instantaneous indicators of Safety Analyses: the reliability $R(t)$, the availability $A(t)$, the maintainability $M(t)$, and
- the average values: Mean Time To Failure (MTTF), Mean Time To Repair (MTTR).

The size of the Markov chain grows, in general, exponentially with the number of components of the system under study. Some tools can deal with Markov chains with over than 1 million states [89]. Nevertheless, the method is applicable only for quite small systems.

Stochastic simulation The principle of stochastic (Monte-Carlo) simulation [2] is to run many histories by drawing at pseudo-random the delays of the transitions and to make statistics on these histories in order to evaluate different performance and reliability indicators. The first advantage of this method is that different probability distributions (not only exponential) are accepted. The second one is the fact that large scale models can be processed using stochastic simulation. The only limit is the execution time of the program because many histories should be simulated to get acceptable results.

Generation of critical sequences of events A critical sequence is a sequence of events leading from the initial state to a critical state. In the case of dynamic models, the order of occurrences of events is important and thus the approximation consisting in extracting minimal cutsets is not suitable: minimal or most probable sequences or sequences of a given length (also called order) can be extracted by simulation of the model.

Total has developed GRIF, a system analysis software platform used to calculate system reliability, availability, performance and safety indicators. It includes several graphical modeling modules (Reliability Block Diagrams, Fault Trees, Markov graphs and Generalized Stochastic Petri Nets) and classical RAMS assessment tools together with a Monte-Carlo simulation engine.

Advantages and drawbacks

Despite their usefulness to represent dynamic models, Markov Chains become rapidly unmanageable because of the exponential explosion of the number of states. Its graphical representation is convenient for small systems and becomes intractable for large scale models. States of the system are represented in an explicit way. The formalism is not compositional (i.e. it does not allow the description of systems as hierarchies of (reusable) components), and it is difficult to represent flow propagation like in Reliability Block Diagrams.

Generalized Stochastic Petri Nets also enable to represent dynamic models. They have a convenient graphical representation but this representation becomes unreadable for large scale models. They are compositional but with some limits. First, it is only possible to represent simple synchronization of events by fusion of transitions. Second, it is quite difficult to represent the propagation of flows. The modeling should provide mechanisms to describe the inputs, the outputs and the relationships amongst them. We can imagine to introduce input and output places and to use the fusion of places [58]. However, it has its limitations.

In [100] J.-P Signoret describes Generalized Stochastic Petri Nets with predicates and messages to be able to represent the propagation of flows easily.

The advantages and drawbacks of classical States/Transitions formalisms are summarized in Table 1.2.

Formalism	**Advantages**	**Drawbacks**
Markov chains	Event-based Graphical representation Dynamic and static models	Lack of structure Difficult to represent remote interactions Only quite small systems
Generalized Stochastic Petri Nets	Event-based Compositional Graphical representation Dynamic and static models	Lack of structure Difficult to represent remote interactions

Table 1.2: States/Transitions formalisms

1.2.3 Extensions of classical formalisms for Safety Analysis

In Safety Analyses, we can use either Boolean formalisms or States/Transitions formalisms. A comparison between these two categories of modeling formalisms is summarized in Figure 1.7. On the one hand, Boolean formalisms are hierarchical, enable to represent easily remote interactions and have

convenient graphical representations but they cannot represent dynamic models. On the other hand, States/Transitions formalisms are more expressive but suffer from a lack of structure and difficulties to represent remote interactions.

As a consequence, many proposals to extend classical (Boolean) formalisms in order to improve their expressive power, to be able to capture dependencies between events, can be found in the literature. In this section we present some of them.

Formalisms / Properties	Boolean		States/Transitions	
	FT	RBD	MC	GSPN
Expressiveness	Static models		Static and Dynamic models	
Event-centric	Yes	Yes	Yes	Yes
Structure	Yes	Yes	No	No
Remote interactions	Yes	Yes	Difficult	Difficult
Graphical representation	Convenient		Convenient for small models	
Assessment	Fault Tree Analysis		Markov Analysis, Stochastic simulation	

Figure 1.7: Comparison of classical formalisms for Safety Analysis

Dynamic Fault Trees (DFT)

Dynamic Fault Trees (DFT) [102] extend traditional (called static) Fault Trees to be able to represent the failure modes which depend on the order of components failures. They introduce different new gates:

PAND Gate: A Priority-AND gate has been introduced to model sequences of failures. The output event of this gate occurs if the input events occur in a specified order.

Spare Gate: A spare gate models components in redundancy. It comprises a main input event and one or more spare input events. When the main input fails, a spare is passed from the standby state to the active state, replacing the main input in its function. A spare gate fails when the main input fails and all spare components are failed or unavailable. A spare component may fail in both the standby and the active states. However, during the standby state, the failure rate is reduced by a factor α.

SEQ Gate: The Sequence enforcing gate forces its events to occur in a specific order.

FDEP Gate: A Functional Dependency gate consists of a trigger event and a set of dependent basic events. The occurrence of the trigger event causes the failure of the basic events.

Dynamic Fault Trees can be assessed by automatic conversion to Markov chains. However, solving a Markov model is much more time and memory consuming than solving a standard Fault Tree. The Markov models can grow exponentially with the number of components used in the modeled systems.

An alternative approach is to use stochastic simulation. The advantages and drawbacks of this method have been discussed earlier.

In [64], the DIFTree methodology has been introduced: it combines solution techniques based on Markov chains, Binary Decision Diagrams (BDD) and simulation.

The Galileo [30, 102] software supports Dynamic and Static Fault Trees and provides a set of assessment tools including a Monte-Carlo simulation engine that uses variance reduction techniques for the analysis of reliable systems.

Boolean logic Driven Markov Processes (BDMP)

Boolean logic Driven Markov Processes (BDMP) [16] is a formalism combining concepts inherited from Fault Trees and Markov chains. Informally speaking, a BDMP is a Fault Tree, where

- leaves (basic events) are represented by Markov processes, and
- gates may trigger other gates (which is used to represent cold redundancies).

The overall tree describes a Markov process. BDMP are implemented as a Library of KB3 workbench, developed by EDF R&D [15]. In practice, BDMP are assessed by generation of most probable sequences of events (performed by FIGSEQ tool integrated in KB3 workbench).

GO-FLOW

GO-FLOW [66] was introduced as an oriented systems analysis technique. This methodology consists of constructing a GO-FLOW chart, by using a set of logical and transition operators connected together to identify the inputs and the outputs of the operators. These connections can represent any physical variable, time or any information. A procedure, to include common cause failures with uncertainty, was introduced [67]. GO-FLOW has been applied to a wide variety of systems in Japan, ranging from the marine reactor MRX [68] to the Shinkansen (Bullet Train) [69].

1.3 Model-Based approach for Safety Analysis

Nowadays, traditional risk assessment methods (Fault Tree Analysis, Event Trees Analysis), presented in Section 1.2 of this chapter, have reached their limits. They rely on too low level modeling formalisms. As a consequence, there is always a gap between the system specifications and the associated safety models. This gap makes safety models hard to design, to share amongst stakeholders and to maintain through the life cycle of systems. Even a minor change in the specification may require a total revisiting of the safety model, which is both time consuming and error prone.

Model-Based Safety Assessment – the reliability engineering branch of Model-Based System Engineering – focuses more and more worldwide attention. The idea is to write models in high level modeling formalisms so as to keep them close to the functional and physical architecture of the system [55]. The high level model can be assessed directly or by its compilation into a low level model, e.g. Fault Tree or Markov chain.

1.3.1 Advantages of Model-Based approach

Compared to classical approaches such as Fault Tree Analysis, Model-Based Safety Assessment presents many advantages:

- Safety models are kept close to functional and physical architectures of the systems under study. Therefore, it is much easier to propagate changes in system specifications as well as to trace changes in safety models.
- Safety models are structurally close to models designed by other system engineering disciplines (system architecture, dynamic system modeling, etc.). This proximity is of a great help to better integrate Safety Analyses with other system design processes.

- Models can be graphically animated. The incident or accident scenarios can be visualized and discussed. In a word, high level models are much easier to share amongst the different stakeholders than lower level ones.

- In general high level modeling languages have a greater expressive power than Boolean formalisms such as Fault Trees or Reliability Blocks Diagrams. It is therefore possible to capture phenomena, such as spare redundancies, shared components, etc.

- High level modeling favors the reuse of models at the component level (via the design of libraries) and at the system level (via the adaptation of a model designed for a project to another project). Experience shows that this is a great source of cost saving.

- For the same reasons, high level modeling favors knowledge capitalization.

1.3.2 Prerequisites for a high level modeling language for Safety Analyses

In this section we discuss different properties that a high level modeling language for Safety Analyses should have.

First of all, a high level modeling language should be formal, i.e. its semantics should be formally defined. In order to be assessed (compiled into a low level formalism, e.g. Fault Tree or Markov chain, simulated, etc.) the model interpretation must not be ambiguous. The formal semantics ensures the correctness of the obtained results.

Second, a high level modeling language should combine the advantages of both Boolean and States/Transitions formalisms:

- It should be event-based. The goal of Safety and Reliability studies is to determine the most probable failure scenarios, i.e. sequences of events leading the system from its nominal state to a failure state (an incident or an accident). As it was seen in Section 1.2 all classical formalisms are event-based. It should be possible to associate probability distributions to events.

- It should have the expressive power of at least States/Transitions formalisms to be able to represent dynamic models.

- At the same time the language must be compositional in order to make it possible to describe systems as hierarchies of components, like in Fault Trees and Reliability Block Diagrams.

- For any reasonable size system, the number of reachable states is just astronomical. It is impossible to represent all the states the system may reach. Therefore, the state space should be represented in an implicit way.

- Fault Trees and Reliability Block Diagrams make it possible to represent instant remote interactions between components of the system under study. The language should make it easy to assemble components “in a Lego way” and to represent the propagation of flows through the system.

Third, the language should be textual but it must be possible to associate different graphical representations with textual models. From our point of view, it is not possible to have a unique graphical representation of the whole model due to its expressiveness: it would be just unreadable. Diagrams like Markov Chains or Generalized Stochastic Petri Nets are very convenient for small systems but their interest is lost in case of industrial scale systems. Graphical representations must be seen as partial views on the whole model, which in practice can be very complex.

Finally, the language must favor models reuse and knowledge capitalization.

1.3.3 High level formalisms for Safety Analysis

Different high level modeling formalisms have been proposed to perform Model-Based Safety Assessment. As suggested in [60], these formalisms can be classified according to the engineering semantics of components interfaces, as follows:

- Failure logic modeling formalisms;
- Failure effects modeling formalisms;
- Hybrid approaches.

In the following section we introduce some high level modeling formalisms dedicated to Safety Analysis.

Some of them are extension of formalisms widely used in other system engineering domains to perform Safety and Reliability Analyses. The system model is expressed in a dedicated formalism (e.g. Matlab/SIMULINK, SysML, AADL, etc.). It is then annotated with reliability data and converted into a low level formalism for Safety and Reliability Analyses (e.g. Fault Trees, Markov chain, etc.).

Some other formalisms have been especially created to represent system failures and perform Safety and Reliability studies.

Hip-HOPS

Based on a structural system model, Hip-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies) [78] is a Safety Analysis technique for automatic generation of Fault Trees and FMEA tables. It describes the structure of the system, in which the basic elements are the system components. Components can be connected via input and output ports which model the Data-Flow through the system. The failure behavior is specified as the failure of system components, failure effects can then propagate along the defined connections to other components. Models can be imported from different modeling tools: Matlab/SIMULINK, Eclipse-based UML tools or SimulationX [76].

In [106], M. Walker and Y. Papadopoulos propose to generate Temporal Fault Trees from Hip-HOPS models. The Hip-HOPS approach has been used to perform architecture optimisation [76] and to develop an automatic, optimal SIL allocation for the automotive domain [77].

AADL Error model

AADL (Architecture Analysis & Design Language) [34], developed and standardized by SAE (Society of Automotive Engineers), is used to model embedded real-time systems. AADL can be used to model both software and hardware components and represent system models as a hierarchy of interconnected components. An Error Model annex [35] has been recently added to AADL specification.

In [95], A.E. Rugina et al. propose to transform AADL Error models into GSPN in order to evaluate dependability indicators. An algorithm to automatically generate Fault Trees from AADL Error models is described in [54].

FIGARO

Developed by EDF R&D, Figaro [17] is a textual modeling language dedicated to dependability assessment of complex systems. It combines object-orientation language features (e.g. inheritance and hierarchical representation) and first order production rules: interaction rules to model the propagation of instantaneous effects and occurrence rules, yielding a list of events that may happen in a state of the system and have a particular semantics related to time.

Figaro is used as a description language to create knowledge bases (i.e. libraries of reusable components) for KB3 [15], a workbench developed by EDF R&D to automatically perform systems

dependability assessment, including Monte-Carlo simulation, Markov chain generation and quantification and generation of critical sequences.

SAML

SAML (Safety Analysis Modeling Language) [45], is a formal synchronous language. A SAML model is expressed in terms of finite stochastic state automata. Automata are all executed in discrete time steps with parallel composition. The semantics of a SAML model is defined as Markov decision process. S3E is a design and verification environment focused on SAML models. It provides a model editor and model analysis tools: a stepwise simulator and translators to the input languages of the probabilistic model checker PRISM and the symbolic model checker NuSMV.

More information about SAML can be found in appendix B, where we compare SAML with AltaRica.

UML, SysML profiles

Constructed as a subset of UML 2.0, SysML is a modeling language for system engineering applications. UML and SysML are graphical modeling languages. They use different types of diagrams to model systems. UML and SysML are notations rather than formal languages.

In [27], P. David et al. propose a method to unify and enhance the development of safety critical systems by linking functional design phase, using SysML, with commonly used reliability techniques (i.e. FMEA and dysfunctional models construction in AltaRica Data Flow). In [109], a framework to automatically generate static Fault Trees from system models specified with SysML is described. Many translations have been defined from specialized UML models to Petri Nets or Fault Trees [12, 70, 61] in order to evaluate system performance or reliability.

1.4 AltaRica

AltaRica is a high level formal modeling language dedicated to Safety Analysis. The first version of the language has been created at the Computer Science Laboratory of Bordeaux (LaBRI) at the end of the nineties [80, 7]. AltaRica is an event-centric language because the primary objective of Safety and Reliability studies is to detect and quantify the most probable sequences of events (failures) leading the system from a nominal state to a degraded state (accident). Deterministic or stochastic delays can be associated with events in order to obtain (stochastic) timed models. In AltaRica, the behavior of components is described by means of state machines. The state of a component is represented by variables (so-called state variables) and their values. The changes of state are possible when, and only when, an event occurs. The occurrence of an event updates the values of the variables.

AltaRica distinguished two types of variables:

- State variables that can be modified only through the firing of transitions;
- Flow variables whose values are calculated from those of state variables thanks to a mechanism described by means of so-called assertions. The assertion is executed after each transition firing.

Flow variables are used to model information circulating between components of a model, i.e. eventually to model remote interactions between these components. This ability to model remote interactions is especially important for Safety Analyses, where one of the primary objectives is to study the consequences of failures of individual components into the system as whole. Widely used Fault Trees and Reliability Block Diagrams rely almost exclusively on this ability.

The behavior of components is described by nodes (also called classes). Components can be assembled into hierarchies, their inputs and outputs can be connected and their transitions can be

synchronized. Models of components can be stored in libraries, what favors the reuse of models and the capitalization of knowledge.

AltaRica is an asynchronous language: only one transition can be fired at a time. However, it offers a versatile mechanism to synchronize events. This mechanism is also useful to represent remote interactions. Common cause failures, shared repair crews, broadcast, etc. can be represented by means of synchronizations.

The semantics of the first version of AltaRica is defined in terms of Constraint Automata [7]. It is still developed by LaBRI's team. AltaRica studio is a workbench developed by LaBRI that supports the modeling and the assessment of this version of AltaRica. Several assessment tools have been developed, such as model-checkers [44], Fault Tree compilers and generators of critical sequences of events [43].

This first version was too resource consuming for industrial scale systems. To be able to assess industrial scale systems, a second version, AltaRica Data-Flow, has been created at IML (Marseilles) in 2002 [88, 14]. Its semantics is based on Mode Automata [88]. In this version only Data-Flow assignments are allowed in the assertion, what made it possible to develop a set of efficient assessment tools such as, Fault Tree compilers [88], generators of critical sequences of events, compilers to Markov chains [89], stochastic [58] and stepwise simulators.

AltaRica Data-Flow is now the core language of several industrial, commercially distributed Integrated Modeling and Simulation Environments:

- Cecilia OCAS (Dassault Aviation),
- Simfia v2 (EADS Apsys), and
- Safety Designer(Dassault Systemes).

These environments make it possible to create, to edit and to simulate models graphically. AltaRica Data-Flow Integrated Modeling and Simulation Environments are widely used across various industries. Many successful industrial applications have been reported in the literature (see for example [10, 97, 13, 11, 52, 85, 98, 23, 4]). The Flight Control System of Dassault Aviation Falcon 7x aircraft have been certified on the basis of AltaRica Data-Flow models [5]. AltaRica Data-Flow models have been used to assess the average production of plants in presence of aleas (e.g. unavailability of machines or human operators) [14].

In 2011, an initiative was launched to standardize the syntax of AltaRica Data-Flow [92]. At the same time a decision was made to change the syntax of the language to make it closer to Modelica [37].

AltaRica is the subject of several PhD thesis. In her PhD thesis [74], C. Pagetti proposes a Timed and a Hybrid extensions of AltaRica to model real time systems, i.e. systems dealing with time constraints. In his PhD thesis, C. Kehren [56] studies architecture modeling patterns to perform Safety Analysis. P. David, in his PhD thesis [26], describes an algorithm to automatically generate AltaRica Data-Flow models from SysML diagrams and FMEA tables. L. Sagaspe, in his PhD thesis [96], uses AltaRica modeling language for reliability allocation in avionic systems.

1.4.1 Assessment tools

A high level modeling language cannot be separated from its assessment tools. Different assessment tools have been created for AltaRica models (see Figure 1.8):

- compilers to Fault Trees [88],
- compilers to Markov chains [89],
- generators of critical sequences,
- stochastic simulators [58],

- stepwise simulators, and
- model checkers [44].

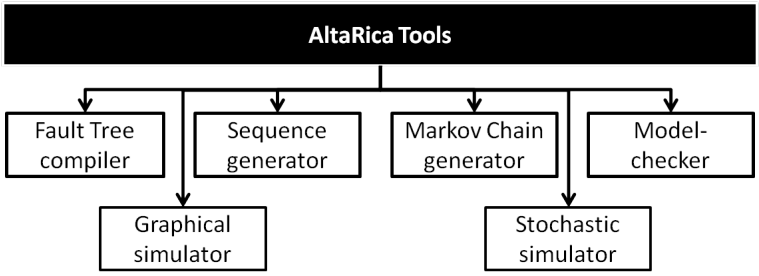


Figure 1.8: AltaRica Tools

Fault Tree compiler

Fault Trees are widely used to perform Safety Analyses and some regulation authorities require to use them to support the certification process. Since high level modeling greatly improves the design, the sharing and the maintenance of models, it is of interest to use them to automatically generate Fault Trees. In many cases high level models can be efficiently compiled into Fault Trees. The generated Fault Tree can be then assessed with calculation engines, such as XFTA [91], in order to calculate minimal cutsets, probabilities of failures, importance factors and other reliability indicators.

In [88], A. Rauzy describes an algorithm to compile AltaRica Data-Flow models into Fault Trees. The proposed algorithm is very efficient on simple but very huge models. Compilers based on this algorithm are integrated in Cecilia OCAS and Safety Designer.

In [43], the authors adapted the algorithm of [88] to the first version of AltaRica (called here AltaRica LaBRI). The compiler is integrated into AltaRica Studio.

Generator of critical sequences of events

A critical sequence is a sequence of events leading from the initial state to a critical state. In some cases, the order of occurrences of events is important and thus the approximation consisting in extracting minimal cutsets (through a compilation of the model into a Fault Tree) is not suitable. In that case, minimal sequences can be extracted.

In [43], authors propose algorithms to generate minimal sequences from AltaRica (LaBRI) models. The generator of critical sequences is integrated into AltaRica Studio.

Markov chain generator

The semantics of AltaRica is a labeled Kripke structure (a reachability graph) that can be interpreted as a Continuous-Time Markov chain, under the condition that all the transitions are either with exponential delays or immediate. Immediate transitions are just collapsed using the fact that an exponential delay with rate λ followed by an immediate transition of probability p is equivalent to a transition with an exponential delay of rate $p\lambda$. The problem of such a compilation is indeed the combinatorial explosion of the number of states and transitions.

The generated Markov chain can be then assessed in order to calculate reliability indicators. Assessment algorithms proposed in [89] deal with Markov chains containing up to 1 million states.

Stepwise simulator

Stepwise simulator enables to perform an interactive step by step simulation of the model. This interactive tool can be very useful to debug models, to play different failure scenarios, etc. The stepwise simulator can be coupled with a graphical simulator as illustrated in [79]. Graphical simulation of models can be used to perform virtual experiments on systems, via models, helping to better understand the system behavior.

Graphical simulators are integrated in all Integrated Modeling and Simulation Environments for AltaRica Data-Flow.

Stochastic simulator

Stochastic (Monte-Carlo) simulation is used when other assessment methods fail. The principle is to run many histories by drawing at pseudo-random the delays of the transitions and to make statistics on these histories. Two types of observers can be defined to calculate the reliability indicators:

- observers on formulas (e.g. the average number of times a formula takes a given value),
- observers on events (e.g. the average number of times an event has been fired).

The only limit of stochastic simulation is the number of histories and the length of histories that are necessary to stabilize the measures.

In his PhD thesis [58], M.-T. Khuu proposes some compilation techniques for AltaRica Data-Flow models in order to improve the efficiency of stochastic simulation.

Model-checker

Model-checking is applied to AltaRica models in two ways:

- The first one consists in designing dedicated model-checkers.
- The second one consists in translating models into the input language of a model-checker.

Model-checking can be used for two reasons:

- To check the temporal properties of the system.
- To check the validity of the model, e.g. to check that all transitions are fireable through some path from the initial state.

Two model-checkers have been designed for AltaRica (LaBRI). The first one is MEC V [44], a symbolic model-checker designed by A. Vincent. The second one is ARC and is an extension of MEC V.

In [19], the authors propose to transform AltaRica Data-Flow models into NuSMV input format.

1.4.2 AltaRica 3.0

AltaRica Data-Flow has now reached an industrial maturity. However, more than ten years of experience showed that both the language and the assessment tools can be improved. To improve the language, we adopted an original approach which consists in viewing a modeling language as a combination of:

- an underlying mathematical formalism: algebraic and ordinary differential equations for Modelica [37], Mealy machines for Lustre [47], Guarded Transitions Systems [90, 84] for AltaRica 3.0, and

	AltaRica (LaBRI)	**AltaRica Data-Flow**
Integrated Modeling and Simulation Environment	AltaRica Studio	Cecilia OCAS Safety Designer Simfia v2
Assessment tools	Model-checkers Fault Tree compiler Sequence generator	Graphical simulator Fault Tree compiler Sequence generator Stochastic simulator

Table 1.3: AltaRica Tools

- a paradigm to structure models: the functional paradigm as in Lucid Synchrone [24], the object-oriented paradigm as in Modelica [37] or the prototype oriented paradigm as in the programming language SELF [73] (or more recently Javascript).

To a large extent, the choice of the underlying mathematical formalism and the structuring paradigm are independent.

The new version of AltaRica, the so-called AltaRica 3.0, improves AltaRica Data-Flow into two directions:

1. First, the new underlying mathematical model – Guarded Transition Systems [90, 84] – makes it possible to handle systems with instant loops and to define acausal components (components for which the input and output flows are decided at run time). It is the subject of the next chapter.
2. Second, the language provides new constructs to structure models. These new constructs makes it possible to match better with cognitive processes of engineers. They are presented in the third chapter of this thesis.

AltaRica 3.0 modeling language is, in fact, the combination of its underlying mathematical formalism, Guarded Transition Systems (GTS), and the paradigm to structure models, System Structure Modeling Language (S2ML):

$$\text{AltaRica 3.0} = \text{S2ML} + \text{GTS}$$

In summary, AltaRica 3.0 models are made of:

- A set of *domains*, user-defined enumerations to represent types of variables;
- A set of *records*, user-defined composed types;
- A set of *functions*, used to group instructions together so that they may be easily reused;
- A set of *classes* or *blocks*, the structural constructs which are presented in details in chapter 3.

The behavior of each individual component, represented by a *class* or a *block*, is modeled by means of:

- *Variables*. State variables are used to represent the component state, they have the attribute *init*. Flow variables represent flows of matter circulating through the component, they are introduced by the attribute *reset*.
- *Parameters*. They are often used to represent failure or repair rates of the components.

- *Events.* They represent failures, repairs, etc. of components. Attributes can be associated with events, e.g. *delay* or *expectation.*
- *Observers.* They are quantities to be observed by the assessment tools.
- *Transitions.* They are used to represent how the component changes its state, if an event occurs.
- *Assertion.* It is an instruction that is used to calculate the value of flow variables after each transition firing.

All these concepts will be presented in details in the next chapters of this thesis.

Summary

In this chapter, we have introduced the basic notions of Safety and Reliability studies. We gave an overview of classical formalisms used to perform Safety Analyses. We also discussed the Model-Based approach for Safety Assessment and introduced some related high level modeling languages. Finally, we presented AltaRica – a high level modeling language dedicated to Safety Analyses.

In the next chapter we will inroduce Guarded Transition Systems, the underlying mathematical formalism of AltaRica 3.0, the new version of AltaRica.

Chapter 2

Guarded Transition Systems (GTS)

High level modeling languages can be seen as a combination of an underlying mathematical formalism and a paradigm to structure models. This chapter presents the underlying mathematical model of the AltaRica 3.0 modeling language – Guarded Transition Systems (GTS). GTS [90] is a states/transitions formalism dedicated to Safety Analyses that generalizes Reliability Block Diagrams, Markov chains and Stochastic Petri nets.

Three operations (free product, connection, synchronization) are defined on Guarded Transition Systems. These operations make it possible to assemble them into hierarchies. Guarded Transition Systems are thus a complete description language (conversely to flat formalisms like Finite State Machines or regular Petri nets).

By introducing a fixpoint mechanism to stabilize the values of flow variables after each transition firing, GTS makes it possible to represent systems with instant loops (for example network systems or electrical systems) and to design acausal components, i.e. components in which the direction of the flow propagation is determined at run time. Thus, GTS also generalizes Mode automata [88] – the underlying mathematical model of AltaRica Data-Flow.

AltaRica 3.0 can just be seen as a convenient way to describe and to structure Guarded Transition Systems.

This chapter is organized as follows. Section 2.1 presents a red wire example of this chapter and discusses its modeling issues. Section 2.2 gives an informal presentation of GTS using examples and discusses its properties. Section 2.3 is dedicated to the formal definition of Guarded Transition Systems. Section 2.4 introduces three operations to compose GTS. Section 2.5 defines the semantics of Guarded Transition Systems. Section 2.6 discusses the representation of flow propagation. Section 2.7 presents Timed and Stochastic extensions of Guarded Transition Systems. Finally, section 2.8 summarizes the properties of formalisms dedicated to Safety Analyses and compares Guarded Transition System with classical formalisms for Safety Analyses.

2.1 Motivations

Example 2.1 (An irrigation system)**.** Consider an irrigation system, depicted Figure 2.1, made of

- Two pumps P1 and P2 supplying water from the rivers R1 and R2 to the fields F1, F2 and F3;
- Three valves V1, V2 and V3.

Both pumps may fail in operation. In normal mode the field F1 is irrigated by the pump P1 and the field F2 is irrigated by the pump P2. The field F3 can be irrigated by both pumps depending on which valve is open V1 or V2 (in Figure 2.1 the valve V1 is open and the valve V2 is closed). If the pump P1 fails, the Field F1 can be irrigated by the pump P2 via the valve V3 if it is open or via the

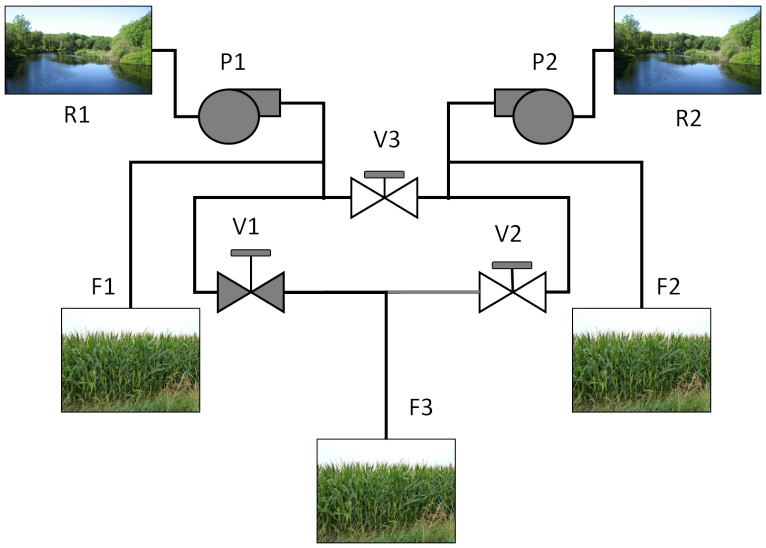


Figure 2.1: An irrigation system

valves V2 and V1 if both are open. In the same way the Field F2 can be irrigated by the pump P1 if the pump P2 is failed.

Valves can also fail in operation: they can be stuck open or stuck closed. Moreover, assume that the pumps may have a common cause failure. To simplify, consider that both pumps are always supplied in water.

The goal of this system is to ensure the irrigation of all of the three fields F1, F2 and F3. □

The example given above encompasses several modeling issues:

- First of all, it contains bidirectional connections between components. Indeed, connections between the valves are bidirectional. For example, the flow can circulate from the valve V1 to the valve V2 and vice-versa from V2 to V1, depending on the global state of the system.
- Second, the system contains an instant feedback loop: when all of the three valves are open, the flow can circulate from the valve V3 to the valve V2, then from V2 to V1 and then go back from V1 to V3. In that case, for example, the left stream of the valve V3 depends instantaneously on its right stream (without the firing of any transition). Electrical systems or computer networks are typical examples of systems for which it is very hard to avoid to introduce loops in models. Difficulties of modeling such systems with instant loops are discussed in [90].

We shall use this system as a red wire of this chapter in order to illustrate different concepts of Guarded Transitions Systems. We show how it is possible to model acausal components and handle looped systems with GTS. We also explain here why some mechanisms with at least the expressive power of fixpoints needs to be introduced in order to deal with looped systems and why fixpoint provides a minimal solution for that purpose.

2.2 Informal presentation

A Guarded Transition System (GTS) is an automaton where states are represented by variable assignments, i.e. variables and their values. Changes of states are represented by transitions triggered by events. It is also possible to represent flows circulating through a network and to synchronize events in order to describe remote interactions between components of the system under study. GTS generalizes both Reliability Block Diagrams, Markov chains and Petri nets.

In this section we present different concepts of GTS. We use AltaRica 3.0 syntax to describe GTS. Note that other syntax can be used to represent GTS.

2.2.1 Data-Flow components

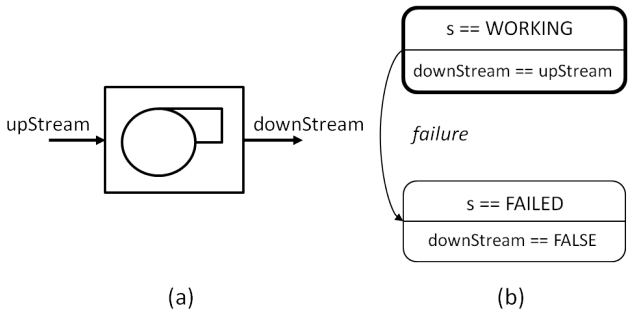


Figure 2.2: A Data-Flow pump

Consider a pump depicted Figure 2.2(a). Assume that it can be either working or failed. In the initial state the pump is working. When an event failure occurs, the pump changes its state to failed. Also, if the pump is working its output flow, represented by the variable `downStream`, is equal to its input flow, represented by the variable `upStream`. Otherwise, it is null.

Such a pump can be represented by a Guarded Transition System in the following way. (See Figure 2.3.)

```
domain ComponentState { WORKING, FAILED }

class DataFlowPump
  ComponentState s (init = WORKING);
  Boolean upStream, downStream(reset = FALSE);
  event failure;
transition
  failure: s == WORKING -> s := FAILED;
assertion
  downStream := if s == WORKING then upStream else FALSE;
end
```

Figure 2.3: GTS representing a Data-Flow pump

The variable `s` represents the state of the pump. It takes its value into the domain `ComponentState`. Its initial value is given by the attribute `init` and is equal to `WORKING`. Its value is modified by the post-condition of the transition labeled by the event `failure`. State variables can be modified only in the post-condition (also called the action) of the transition.

Variables `upStream` and `downStream` represent the input flow and the output flow of the pump respectively. They are called flow variables. Their initial or default values are given by the attribute `reset`. They depend on the state variables and can be modified only in the assertion. The assertion stays that if the variable `s` is equal to `WORKING` then the variable `downStream` equals to the variable `upStream`, otherwise it equals to `FALSE`. The value of flow variables is updated after each transition firing.

The transition, labelled by the event `failure`, is fireable only when the value of the state variable `s` is `WORKING`. The firing of the transition first sets the value of `s` to `FAILED`, then it updates the values

of flow variables.

Figure 2.2(b) shows a graphical representation of the GTS that describes the pump. States are represented by rectangles with rounded corners. They are labeled by the state variable and its value. The value of flow variables is given under the separation line. The initial state is marked in bold. Transitions are represented by arrows joining states.

The model of the pump is called a Data-Flow component (see section 2.6 for more details on Data-Flow assertions). It can be represented by a Mode automaton [88].

2.2.2 Acausal components

With Guarded Transition Systems it is also possible to represent acausal components, i.e. components for which inputs and outputs are decided at run time, which is not the case of Mode automata where all components are assumed to be Data-Flow.

In the example of the *Irrigation system*, depicted in Figure 2.1, flows between components are bidirectional. To be able to represent bidirectional flows, we need to define acausal components. As a consequence, we should modify the model of the Data-Flow pump in order to make it acausal.

An acausal pump

The GTS, describing an acausal pump, is given in Figure 2.4.

```
domain ComponentState { WORKING, FAILED }

class Pump
  ComponentState s (init = WORKING);
  Boolean upStream, downStream(reset = FALSE);
  event failure;
transition
  failure: s == WORKING -> s := FAILED;
assertion
  if s == WORKING and upStream then downStream := TRUE;
end
```

Figure 2.4: GTS representing an acausal pump

This model is similar to the previous one: only the assertion is different from the previous model of the pump. The assertion stands that if the pump is working and there is a flow in the upstream then there is a flow in the downstream. Otherwise, nothing can be said about the downstream of the pump. In this last case, the value of the flow variable is set to its default value, given by the attribute `reset`.

A valve

Now consider the valve pictured Figure 2.5(a). It can be either open or closed and it may be stuck (failed). The valve changes from open to closed (respectively from closed to open) if it is not failed and if the event close (respectively open) occurs. It gets stuck when the event failure occurs. If the valve is open, the flow can circulate through the valve. Otherwise, nothing can go through it. In the last case, the value of flow variables is set to their default value.

The GTS representing such a valve is given Figure 2.6.

In this model, it is not specified which flow variable represents the input flow and the output flow of the valve. The assertion stays that if the variable `s` equals to `WORKING` then flow variables

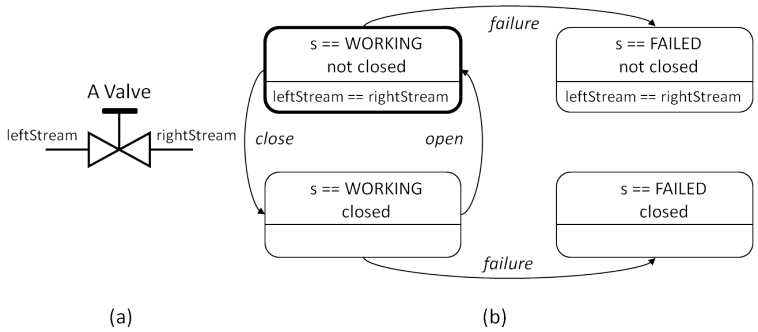


Figure 2.5: A Valve

```
domain ComponentState { WORKING, FAILED }

class Valve
  ComponentState s (init = WORKING);
  Boolean isClosed(init = FALSE);
  Boolean rightStream, leftStream(reset = FALSE);
  event open, close, failure;
transition
  open: s==WORKING and isClosed -> isClosed := FALSE;
  close: s==WORKING and not isClosed -> isClosed := TRUE;
  failure: s == WORKING -> s := FAILED;
assertion
  if s == WORKING then rightStream :=: leftStream;
end
```

Figure 2.6: GTS representing a valve

`leftStream` and `rightStream` are connected together; otherwise there is no relation amongst them. The direction of the flow propagation is determined at run time: either the variable `leftStream` may give value to the variable `rightStream` or vice-versa depending on the global state of the system where the component `Valve` is used. This mechanism will be explained in more details later.

Figure 2.5(b) shows a graphical representation of the GTS that describes the valve.

2.2.3 Hierarchical models

Consider the *Irrigation system* from Example 2.1. The GTS representing a field is given Figure 2.7.

```
class Field
  Boolean inStream(reset = FALSE);
end
```

Figure 2.7: GTS of a Field

Now, the GTS representing the three valves, the two pumps and the three fields have to be combined together to get the GTS that represents the system as a whole. We need to compose GTS, to connect together some variables and to synchronize some events (to represent the common cause

failure of the pumps).

Guarded Transitions Systems can be easily assembled into hierarchies of reusable models of components by means of three operations: the free product, the connection and the synchronization.

Free product

First we need to assemble together the independent Guarded Transition Systems representing the valves, the pumps and the fields. This operation is called a free product and consists in putting together two or more independent GTS. The result of this operation is a Guarded Transition System that contains all variables, events, transitions and assertions of the independent GTS. In the resulting GTS every named object (e.g. variable, event, etc.) is prefixed by the name of the GTS followed by a dot.

Connection

Now, pumps, valves and fields are completely independent. To represent the circulation of water from pumps to fields, we need to connect some flow variables together. The connection consists in compelling one or more variables to be equal to a function of some other variables. It is performed by adding several assertions to the previous model.

Synchronization

As for other states/events formalisms such as Petri nets, transitions of Guarded Transition Systems are assumed to be asynchronous: two transitions cannot be fired simultaneously. Synchronizations are used to compel a set of events to occur simultaneously.

In our example, we shall use the synchronization to represent the common cause failure of the pumps. A new event is defined. It synchronizes the individual failures of the pumps.

The three operations (free product, connection and synchronization) are presented in more details in Section 2.4.

2.3 Formal definition

In order to formally introduce Guarded Transition Systems, we need to define some syntactic constructs. They are described hereafter.

2.3.1 Expressions

Consider a Universe $\mathcal{U}$:

- a denumerable set of constants $\mathcal{C}$,
- a finite set of operators $\mathcal{O}$,
- a function $\alpha : \mathcal{O} \to \mathbb{N}$, that associates an arity to each operator, and
- a standard interpretation, denoted by $[\![.]\!]$, that associates to each operator $op \in \mathcal{O}$ a partial function $[\![op]\!] : \mathcal{C}^{\alpha(op)} \to \mathcal{C}$.

We assume moreover that $\mathcal{C}$ contains at least the Boolean constants $TRUE$ and $FALSE$ and that $\mathcal{O}$ contains at least the Boolean operators "and", "or" and "not" with their usual interpretation.

Let $\mathcal{V}$ be a finite set of symbols, called variables, and let dom be a function that associates with each variable its domain (a set of values of the variable v), i.e. finite or denumerable subset of $\mathcal{C}$.

Definition 2.1 (Expressions)**.** The set of expressions $\mathcal{E}$ is built as the smallest set such that:

- $\forall c \in \mathcal{C}$, c is an expression;
- $\forall v \in \mathcal{V}$, v is an expression;
- if op is an operator and $E_1, \ldots, E_{\alpha(op)}$ are expressions, then $op(E_1, \ldots, E_{\alpha(op)})$ is also an expression.

Let us denote by $var(e) \subseteq \mathcal{V}$ a set of variables occurring in the expression e, formally defined as follows:

- for all constant $c \in \mathcal{C}$, $var(c) = \emptyset$;
- for all variable $v \in \mathcal{V}$, $var(v) = \{v\}$;
- $var(op(E_1, E_2, \ldots, E_{\alpha(op)})) = var(E_1) \cup var(E_2) \cup ... \cup var(E_{\alpha(op)})$.

Definition 2.2 (Variable assignment)**.** A variable assignment is a function $\sigma : \mathcal{V} \to \mathcal{C}$, that associates for each variable $v \in \mathcal{V}$ its value.

We say that the variable assignment σ is acceptable if

$$\forall v \in \mathcal{V},\ \sigma(v) \in dom(v).$$

Variable assignments are lifted-up into functions from $\mathcal{E}$ to $\mathcal{C}$ in the usual way: let σ be an acceptable variable assignment, let c be a constant, let op be an operator, and finally let $E_1, E_2, \ldots, E_{\alpha(op)}$ be expressions, then

$$\begin{aligned} &\sigma(c) = c \\ &\sigma(op(E_1, E_2, \ldots, E_{\alpha(op)})) = [\![op]\!](\sigma(E_1), \sigma(E_2), \ldots, \sigma(E_{\alpha(op)})) \end{aligned} \tag{2.1}$$

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 does not give 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

$$\sigma[c/v](w) = \begin{cases} c & \text{if } w = v, \\ \sigma(w) & \text{for any other variable } w. \end{cases} \tag{2.2}$$

From now, we assume that the set of variables $\mathcal{V}$ is a disjoint union of the set $\mathcal{S}$ of state variables and the set $\mathcal{F}$ of flow variables, $\mathcal{V} = \mathcal{S} \uplus \mathcal{F}$. Moreover, we assume that each variable has a default or initial value.

2.3.2 Instructions

Guarded Transition Systems are built on the following instructions: an empty instruction, an assignment, a conditional assignment and a parallel composition.

Definition 2.3 (Instructions)**.** 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 "$v := E$" is an instruction;
- if C is a Boolean expression, I is an instruction, then "if C then I" is an instruction;
- if I_1 and I_2 are instructions, then so is the parallel composition "$I_1; I_2$".

We shall consider 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 *assertions*.

Example 2.2 (A valve)**.** Consider the GTS of the valve (see Figure 2.5) given Figure 2.6. The assertion is equivalent to the following block of instructions.

```
{if s==WORKING then leftStream := rightStream;
if s==WORKING then rightStream := leftStream;}
```

Only flow variables are assigned in the assertion. This block of instructions is composed of two conditional assignments. □

2.3.3 Definition

Definition 2.4 (Guarded Transition System)**.** A Guarded Transition System is a quintuple

$$G = \langle V, E, T, A, \iota \rangle,$$

where:

- V is a set of variables, divided into two disjoint sets S of state variables and F of flow variables: $V = S \uplus F$;
- E is a set of symbols, called events;
- T is a set of transitions, i.e. of triples $\langle e, G, P \rangle$ also denoted $e : G \to P$, where e is an event of E, G is a guard, a Boolean formula built over the set V, and P is an instruction built on variables of V, called an action or a post-condition;
- A is an assertion, i.e. an instruction built on variables of V;
- ι is an assignment of variables of V, so-called an initial or default assignment.

A transition $e : G \to P$ is fireable in a given state σ, i.e. for a given variable assignment σ, if

$$\sigma(G) = TRUE.$$

The firing of transition $e : G \to P$ is performed into two steps: first, state variables are updated by applying the instruction P to σ; second, flow variables are propagated using the assertion A. This mechanism is described in Section 2.5.

Example 2.3 (A valve)**.** Consider again the valve given Figure 2.5. The GTS $G = \langle V, E, T, A, \iota \rangle$ that describes the valve is formally defined as follows.

- The set of variables $V = S \uplus F$ is the disjoint set of
  - the state variables $S = \{s; isClosed\}$ with
    $$dom(s) = \{WORKING; FAILED\}$$
    $$dom(isClosed) = \{TRUE; FALSE\},$$
    and
  - the flow variables $F = \{leftStream; rightStream\}$ with
    $$dom(lefStream) = dom(rightStream) = \{TRUE; FALSE\}$$
- The set of events E contains three events $E = \{open; close; failure\}$.
- T contains the following transitions:
  $open : isClosed \ and \ s == WORKING \rightarrow isClosed := FALSE$
  $close : not \ isClosed \ and \ s == WORKING \rightarrow isClosed := TRUE$
  $failure : s == WORKING \rightarrow s := FAILED$
- The assertion A is a block of instructions that contains two conditional assignments:
  $if \ s == WORKING \ then \ leftStream := rightStream$
  $if \ s == WORKING \ then \ rightStream := leftStream$
- Finally, the initial or default variable assignment ι is as follows:
  $(s = WORKING, isClosed = FALSE, leftStream = FALSE, rightStream = FALSE)$.

□

2.4 Composition of GTS

A major prerequisite for a high level description language is to be compositional, i.e. to allow the description of systems as hierarchies of (reusable) components. Guarded Transition Systems can be assembled by means of three operations: free product, connection and synchronization. These operations produce a Guarded Transition System: any hierarchy can be flattened into an equivalent Guarded Transition System.

2.4.1 Free product

To build hierarchies, we need an operation that groups together several GTS. The first operation defined to assemble two GTS is the free product.

Definition 2.5 (Independence of two GTS)**.** Let $G_1 = \langle V_1, E_1, T_1, A_1, \iota_1 \rangle$ and $G_2 = \langle V_2, E_2, T_2, A_2, \iota_2 \rangle$ be two GTS. G_1 and G_2 are said independent if they are built over distinct sets of variables and events, i.e.

1. $V_1 \cap V_2 = \emptyset$;
2. $E_1 \cap E_2 = \emptyset$;

Definition 2.6 (Free product)**.** Let $G_1 = \langle V_1, E_1, T_1, A_1, \iota_1 \rangle$ and $G_2 = \langle V_2, E_2, T_2, A_2, \iota_2 \rangle$ be two independent GTS. The free product $G = \langle V, E, T, A, \iota \rangle$ of G_1 and G_2, denoted $G_1 \times G_2$ is defined as follows:

- $V = V_1 \cup V_2$;
- $E = E_1 \cup E_2$;
- $T = T_1 \cup T_2$;
- $A = A_1; A_2$, where ; denotes the parallel composition of instructions;
- $\iota = \iota_1 \circ \iota_2$, where $\circ$ denotes the composition of functions.

Note that since the two GTS are assumed to be built over distinct sets of variables and events, the product $\times$ is commutative and associative.

In practice, this operation can be represented as an instantiation of classes representing each independent component.

Example 2.4 (An irrigation system)**.** Consider the irrigation system from Example 2.1. It is made of two pumps, three valves and three fields. The GTS representing the irrigation system is a composition of the Guarded Transition Systems representing the valves, the pumps and the fields:

```
block IrrigationSystem
  Pump P1, P2(upStream.reset = true);
  Valve V1, V2, V3;
  Field F1, F2, F3;
end
```

In concrete terms the block `IrrigationSystem` embeds two instances of the class `Pump`, named `P1` and `P2`, three instances of the class `Valve`, named `V1`, `V2` and `V3`, and three instances of the class `Field`, named `F1`, `F2` and `F3`.

The resulting GTS contains all variables, events, transitions and assertions of the independent Guarded Transition Systems. All named objects, e.g. variables or events, are prefixed by the name of the instance (see Figure 2.9). □

2.4.2 Connection

The connection consists in compelling one or more flow variables to be equal to a function of some other variables. Guarded Transition Systems introduce a special operator to represent acausal connections: "`:=:`".

Definition 2.7 (Acausal connection)**.** Let $G = \langle V = S \uplus F, E, T, A, \iota \rangle$ be a GTS. Let v and w be two flow variables: $v, w \in F$. An(The) acausal connection of flow variables v and w is an(the) instruction of the form $v :=: w$. It is equivalent to the two assignments: $\{v := w; w := v;\}$.

It is important to notice that, acausal connections introduce loops.

In practice, connections are represented by assertions.

Example 2.5 (An irrigation system)**.** In order to represent flows of water circulating from the pumps to the fields, the following connections are added to the previous model of the irrigation system:

```
block IrrigationSystem
  Pump P1, P2(upStream.reset = true);
  Valve V1, V2, V3;
  Field F1, F2, F3;
assertion
  V1.rightStream :=: V2.leftStream;
  V2.rightStream :=: V3.rightStream;
  V3.leftStream :=: V1.leftStream;
  V3.rightStream :=: P2.downStream;
  V3.leftStream :=: P1.downStream;
  F1.inStream := P1.downStream;
  F2.inStream := P2.downStream;
  F3.inStream := V1.rightStream;
end
```

The Guarded Transition Systems are not completely independent anymore. Their flow variables are connected together. □

2.4.3 Synchronization

As in other states/transitions formalisms such as Petri nets, transitions of GTS are assumed to be asynchronous: two transitions cannot be fired simultaneously. The synchronization mechanism consists in compelling a set of events to occur simultaneously.

Definition 2.8 (Synchronization)**.** Let $G = \langle V, E, T, A, \iota \rangle$ be a GTS. A synchronization is a transition in the following form:

$$e : !a_1 \ \& \ \ldots \ \& \ !a_m \ \& \ ?b_1 \ \& \ \ldots \ \& \ ?b_n \ \& \ L_1 \to R_1 \ \& \ \ldots \ \& \ L_r \to R_r,$$

$$m \geq 0, n \geq 0, r \geq 0,$$

where

1. e, $a_i, i = 0..m$, $b_j, j = 0..n$, are events from E;
2. Events are prefixed by either ! or ?, called the modality: ! meaning that the event is mandatory and ? meaning that the event is optional;
3. $L_k \to R_k, k = 0..r$, are unnamed transitions, $L_k, k = 0..r$, are guards (i.e. Boolean expressions built over variables from V), $R_k, k = 0..r$, are actions built over V.

The synchronization defines a set of new transitions. They are calculated in the following way.
First case: $m \geq 1$ or $r \geq 1$
For each set of transitions (there may be several):

$$a_1 : G_1 \to P_1, \ldots, a_m : G_m \to P_m$$

$$b_1 : H_1 \to Q_1, \ldots, b_n : H_n \to Q_n, n \geq 0$$

the following new transition is created:

$$e : G_1 \text{ and } \ldots \text{ and } G_m \text{ and } L_1 \text{ and } \ldots \text{ and } L_r \to$$

$$\{P_1; \ldots; P_m; \text{ if } H_1 \text{ then } Q_1; \ldots; \text{ if } H_n \text{ then } Q_n; R_1; \ldots; R_r\}$$

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

Second case: $m = 0, r = 0, n > 1$
For each set of transitions (there may be several):

$$b_1 : H_1 \to Q_1, \ldots, b_n : H_n \to Q_n$$

the following new transition is created:

$$e : H_1 \text{ or } \ldots \text{ or } H_n \to \{ \text{ if } H_1 \text{ then } Q_1; \ldots; \text{ if } H_n \text{ then } Q_n\}$$

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.

We will see in Section 2.5.1 that all instructions in the action of a transition are executed in parallel. As a result the order in which the instructions are written (and therefore the order in which events appear in the synchronization) is not important. The operation of events synchronization is commutative and associative.

Just as assertions make it possible to link variables from different GTS, synchronizations make it possible to link their events (and therefore transitions).

Transitions involved in a synchronization continue to exist individually. Together with synchronizations it is useful to have a hiding mechanism. If an event is hidden, the transitions labelled with this event are not fireable anymore.

Example 2.6 (An irrigation system)**.** To illustrate the synchronization mechanism, consider again the irrigation system pictured Figure 2.1. We shall use the synchronization to represent the common cause failure of the pumps. To do so, we define a new event `ccf`, and a new transition labelled by this event.

```
block IrrigationSystem
  Pump P1, P2(upStream.reset = true);
  Valve V1, V2, V3;
  Field F1, F2, F3;
  event ccf;
transition
  ccf: ?P1.failure & ?P2.failure;
assertion
  ...
end
```

Figure 2.8: GTS of the irrigation system

The transition `ccf` synchronizes the events `P1.failure` and `P2.failure`. It corresponds to the following flattened transition:

```
ccf: P1.s==WORKING or P2.s==WORKING ->
  {if P1.s==WORKING then P1.s := FAILED;
  if P2.s==WORKING then P2.s := FAILED;}
```

All events involved in the synchronization are optional (modality ?). The synchronized transition is fireable if at least one of the pumps is operational. Synchronized events (and transitions) are not removed from the model. They represent individual failures of the pumps. □

With the operations defined to compose Guarded Transition Systems (free product, connection and synchronization), it is easy to create hierarchical models, i.e. to assemble "on-the-shelf" models of components in a Lego way. Any hierarchical model can be flattened into an equivalent GTS using previously defined rules. As an illustration, the flattened GTS of the irrigation system is given Figure 2.9. The hierarchical model is presented Figure 2.8.

2.5 Semantics

Guarded Transition Systems $G = \langle V, E, T, A, \iota \rangle$ are implicit representations of Kripke structures, i.e. of graphs whose nodes are labeled by variable assignments and whose edges are labeled by events. This graph is called a Reachability graph of G.

To explain how this graph is calculated, we first need to define the semantics of instructions.

2.5.1 Semantics of instructions

As it was mentioned earlier, 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. Assertion is used to calculate flow variables, representing information circulating amongst the components of the model. Their value depends on the value of state variables. It is recalculated after each transition firing.
So, we shall consider 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 *assertions*.

Semantics of actions

Let σ be the variable assignment just before the firing of the transition $t = \langle e, G, P \rangle$. Applying the instruction P to the variable assignment σ consists in calculating a new variable assignment τ according to the rules given in a Structural Operational Semantics style in the Table 2.1.
We start with $\tau = \emptyset$. Then,

- (S0): P is an empty instruction: τ is left unchanged.
- 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.
- 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.
- P is a block of instructions "$\{I_1, \ldots, I_n\}$":
  (S7)-(S12): Instructions I_1, ..., I_n are successively applied to τ. The 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 the execution of I_2. Although tools will probably make a systematic choice, this semantics is independent of this choice.

```
class FlattenIrrigationSystem
  ComponentState V1.s, V2.s, V3.s, P1.s, P2.s(init = WORKING);
  Boolean V3.isClosed, V2.isClosed(init = TRUE);
  Boolean V1.isClosed (init = FALSE);
  Boolean P1.upStream, P2.upStream (reset = TRUE);
  Boolean P1.downStream, P2.downStream (reset = FALSE);
  Boolean F1.inStream, F2.inStream, F3.inStream (reset = FALSE);
  Boolean V1.leftStream, V2.leftStream, V3.leftStream(reset = FALSE);
  Boolean V1.rightStream, V2.rightStream, V3.rightStream(reset = FALSE);
  event V1.open, V2.open, V3.open, V1.close, V2.close, V3.close;
  event V1.failure, V2.failure, V3.failure, P1.failure, P2.failure, ccf;
transition
  V1.open : V1.isClosed and V1.s == WORKING -> V1.isClosed := FALSE;
  V2.open : V2.isClosed and V2.s == WORKING -> V2.isClosed := FALSE;
  V3.open : V3.isClosed and V3.s == WORKING -> V3.isClosed := FALSE;
  V1.close : not V1.isClosed and V1.s == WORKING -> V1.isClosed := TRUE;
  V2.close : not V2.isClosed and V2.s == WORKING -> V2.isClosed := TRUE;
  V3.close : not V3.isClosed and V3.s == WORKING -> V3.isClosed := TRUE;
  V1.failure : V1.s == WORKING -> V1.s := FAILED;
  V2.failure : V2.s == WORKING -> V2.s := FAILED;
  V3.failure : V3.s == WORKING -> V3.s := FAILED;
  P1.failure : P1.s == WORKING -> P1.s := FAILED;
  P2.failure : P2.s == WORKING -> P2.s := FAILED;
  ccf: P1.s==WORKING or P2.s==WORKING ->
    {if P1.s==WORKING then P1.s := FAILED;
    if P2.s==WORKING then P2.s := FAILED;}
assertion
  if P1.s==WORKING and P1.upStream then P1.downStream := TRUE;
  if P2.s==WORKING and P2.upStream then P2.downStream := TRUE;
  if not V1.isClosed then V1.leftStream := V1.rightStream;
  if not V1.isClosed then V1.rightStream := V1.leftStream;
  if not V2.isClosed then V2.leftStream := V2.rightStream;
  if not V2.isClosed then V2.rightStream := V2.leftStream;
  if not V3.isClosed then V3.leftStream := V3.rightStream;
  if not V3.isClosed then V3.rightStream := V3.leftStream;
  V1.rightStream := V2.leftStream;
  V2.leftStream := V1.rightStream;
  V2.rightStream := V3.rightStream;
  V3.rightStream := V2.rightStream;
  V3.leftStream := V1.leftStream;
  V1.leftStream := V3.leftStream;
  V3.rightStream := P2.downStream;
  P2.downStream := V3.rightStream;
  V3.leftStream := P1.downStream;
  P1.downStream := V3.leftStream;
  F1.inStream := P1.downStream;
  F2.inStream := P2.downStream;
  F3.inStream := V1.rightStream;
end
```

Figure 2.9: Flattened GTS of the irrigation system

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

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

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

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

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

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

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

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

Table 2.1: The semantics of actions

It is important to note that in the above mechanism, right hand side of assignments and conditions of conditional instructions are evaluated in the variable assignment 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 to a different value without raising an error. For example, a conditional assignment `if x < 5 then x := x + 1;` is executed in the following way: if $\sigma(x) < 5$, then $\tau(x)$ is set to $\sigma(x) + 1$, otherwise it is left unchanged.
We denote by $Update(P, \sigma)$ a function that:

- Extends a given partial variable assignment τ of S by means of a total variable assignment σ of V from the instruction P according to the rules given in the Table 2.1 and starting with $\tau = \emptyset$.
- Completes τ by setting $\tau(v) = \sigma(v)$ for all variables from S that are not given a value by the previous step.

Example 2.7 (An irrigation system)**.** Consider the transition `ccf` in the model of the irrigation system, depicted Figure 2.1,

```
ccf: P1.s==WORKING or P2.s==WORKING ->
  {if P1.s==WORKING then P1.s := FAILED;
  if P2.s==WORKING then P2.s := FAILED;}
```

and a variable assignment

$$\sigma = (P1.s = FAILED,\ P2.s = WORKING, \ldots)$$

The transition `ccf` is fireable in σ, since its guard is verified. The new variable assignment τ, obtained after the transition firing, is calculated as follows. τ is empty at the beginning.

$$\tau = (P1.s = \emptyset,\ P2.s = \emptyset, \ldots)$$

After the execution of the action, τ is as follows:

$$\tau = (P1.s = \emptyset,\ P2.s = FAILED, \ldots)$$

The variable $P1.s$ has not been given a value. So it receives its value in σ and finally:

$$\tau = (P1.s = FAILED,\ P2.s = FAILED, \ldots)$$

□

Semantics of instructions in assertion

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

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

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 A extends a variable assignment τ which is total on S but possibly partial on F, according to the rules given in the Table 2.2.
Let D be a set of unevaluated flow variables, we start with $D = F$. Then,

- (S0): A is an empty instruction: τ is left unchanged.
- A is an assignment "$v := E$":
  1. $\tau(Exp)$ can be evaluated in τ, i.e. all variables of E have a value in τ.
     (S3): If v already has a value in τ, then if $\tau(v) \neq \tau(E)$ then an error is raised
     (S1): else τ is left unchanged.
     (S2): Otherwise, $\tau(v)$ is set to $\tau(E)$ and v is removed from D.
  2. $\tau(Exp)$ cannot be evaluated in τ, then τ is left unchanged.
- A is a conditional assignment "if C then I":
  1. $\tau(C)$ can be evaluated in τ.
     (S4): If $\tau(C) = TRUE$, then the instruction I is applied to τ.
     (S5): Otherwise, τ is left unchanged.
  2. Otherwise, τ is left unchanged.
- A is a block of instructions "$\{I_1, \ldots, I_n\}$":
  (S7)-(S12): Instructions $I_1, \ldots, I_n$ are repeatedly applied to τ until there is no more possibility to assign a flow variable. This set of rules is non-deterministic but its result does not depend on the execution order.

Let ι be an assignment that associates each variable with its initial or default value. If after applying A to π there are unevaluated variables in D, then all these variables are set to their default values ($\forall v \in D\ \tau(v) = \iota(v)$) and A is applied to τ in order to verify that all assignments are satisfied. If there is at least one assignment that is not satisfied, then an error is raised.

We denote by $Propagate(A, \iota, \tau)$ a function that:

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

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

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

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

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

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

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

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

Table 2.2: The semantics of assertions

- Extends the partial variable assignment τ by the instruction A according to the rules given in the Table 2.2.

- Completes τ by setting all unassigned variables to its default values

$$\forall v \in F : \tau(v) =?/\tau(v) = \iota(v).$$

Example 2.8 (An irrigation system)**.** Consider the irrigation system, depicted Figure 2.1, and the assertion of the corresponding GTS given Figure 2.9.

```
(1) if P1.s==WORKING and P1.upStream then P1.downStream := TRUE;
(2) if P2.s==WORKING and P2.upStream then P2.downStream := TRUE;
(3) if not V1.isClosed then V1.leftStream := V1.rightStream;
(4) if not V1.isClosed then V1.rightStream := V1.leftStream;
(5) if not V2.isClosed then V2.leftStream := V2.rightStream;
(6) if not V2.isClosed then V2.rightStream := V2.leftStream;
(7) if not V3.isClosed then V3.leftStream := V3.rightStream;
(8) if not V3.isClosed then V3.rightStream := V3.leftStream;
(9) V1.rightStream := V2.leftStream;
(10) V2.leftStream := V1.rightStream;
(11) V2.rightStream := V3.rightStream;
(12) V3.rightStream := V2.rightStream;
(13) V3.leftStream := V1.leftStream;
(14) V1.leftStream := V3.leftStream;
(15) V3.rightStream := P2.downStream;
(16) P2.downStream := V3.rightStream;
(17) V3.leftStream := P1.downStream;
(18) P1.downStream := V3.leftStream;
(19) F1.inStream := P1.downStream;
(20) F2.inStream := P2.downStream;
(21) F3.inStream := V1.rightStream;
```

Let us illustrate how this assertion is calculated on a couple of configurations. First, consider a configuration, where both pumps are operational, valves V3 and V2 are closed and V1 is open. In this configuration conditional assignments (5) – (9) cannot be executed because their conditions are not verified. The propagation algorithm works as follows:

From the instruction (1), $P1.downStream \leftarrow true$.
From the instruction (2), $P2.downStream \leftarrow true$.
From the instruction (17), $V3.leftStream \leftarrow true$.
From the instruction (15), $V3.rightStream \leftarrow true$.
From the instruction (19), $F1.inStream \leftarrow true$.
From the instruction (20), $F2.inStream \leftarrow true$.
From the instruction (14), $V1.leftStream \leftarrow true$.
From the instruction (4), $V1.rightStream \leftarrow true$.
From the instruction (21), $F3.inStream \leftarrow true$.
From the instruction (10), $V2.leftStream \leftarrow true$.
From the instruction (11), $V2.rightStream \leftarrow true$.

In this configuration the fields F1 and F3 are irrigated by the pump P1 and the field F2 is irrigated by the pump P2. The flow is propagated from the pump P1 to the valve V1, then from the valve V1 to the valve V2 and to the field F3. In the same way, the flow is propagated from the pump P2 to the valve V2 and to the field F2.

Now consider another configuration, where the pump P1 is failed, the valve V2 is open. In this configuration conditional assignments (1), (7), (8) cannot be evaluated thus their conditions are not verified. The propagation algorithm works as follows:

From the instruction (2), $P2.downStream \leftarrow true$.
From the instruction (15), $V3.rightStream \leftarrow true$.
From the instruction (11), $V2.rightStream \leftarrow true$.

From the instruction (20), $F2.inStream \leftarrow true$.
From the instruction (5), $V2.leftStream \leftarrow true$.
From the instruction (9), $V1.rightStream \leftarrow true$.
From the instruction (21), $F3.inStream \leftarrow true$.
From the instruction (3), $V1.leftStream \leftarrow true$.
From the instruction (13), $V3.leftStream \leftarrow true$.
From the instruction (18), $P1.downStream \leftarrow true$.
From the instruction (19), $F1.inStream \leftarrow true$.

In this configuration the fields F1, F2 and F3 are irrigated by the pump P2: the flow is propagated from P2 to the fields through the valves V2 and V1. □

Let ι be a total assignment that associates for each variable v from V 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 $Fire(P, A, \iota, \sigma)$ an assignment defined as follows:

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

2.5.2 Reachability graph

Guarded Transition 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.

Definition 2.9 (Reachability graph)**.** The semantics of a Guarded Transition System $G = \langle V = S \uplus F, E, T, A, \iota \rangle$ is a Kripke structure, i.e. a graph $\Gamma = (\Sigma, \Theta)$, where

- Σ is a set of variable assignments, also called states (nodes of the graph),
- Θ is a set of triples $\langle s, e, q \rangle$, $s, q \in \Sigma$, $e \in E$ (transitions of the graph).

Γ is the smallest Kripke structure, such that the following is verified:

1. $\sigma_0 = Propagate(A, \iota, \iota) \in \Sigma$. σ_0 is the initial state of the Kripke structure.
2. If $\sigma \in \Sigma$ and $\exists t = \langle e, G, P \rangle \in T$ such that $\sigma(G) = TRUE$, then the state $\tau = Fire(P, A, \iota, \sigma) \in \Sigma$ and the transition $(\sigma, e, \tau) \in \Theta$,

The graph Γ is called the Reachability graph of G.

The calculation of $\Gamma = (\Sigma, \Theta)$ may raise errors. A well-designed Guarded Transition Systems avoids this problem.

Example 2.9 (A pump and a valve connected in series)**.** Consider a part of the irrigation system, depicted Figure 2.1, composed of a pump P1, a valve V1 and a field F1. The Guarded Transition Systems representing a pump, a valve and a field are given in Figures 2.4, 2.6, and 2.7 respectively.

The reachability graph of this system is pictured Figure 2.10. States are represented by rectangles with rounded corners. They are labelled by state variables and their value. For the sake of clarity, the value of only one flow variable is indicated under the separation line: F1.isIrrigated. The initial state is marked in bold. Transitions are represented by arrows joining states and are labelled by events.

□

Definition 2.10 (Free product of reachability graphs)**.** Let G_1 and G_2 be two independent GTS and let $\Gamma_1 = (\Sigma_1, \Theta_1)$ and $\Gamma_2 = (\Sigma_2, \Theta_2)$ be their reachability graphs. The reachability graph $\Gamma = (\Sigma, \Theta)$ is a free product of Γ_1 and Γ_2, denoted $\Gamma_1 \otimes \Gamma_2$, if the following holds:

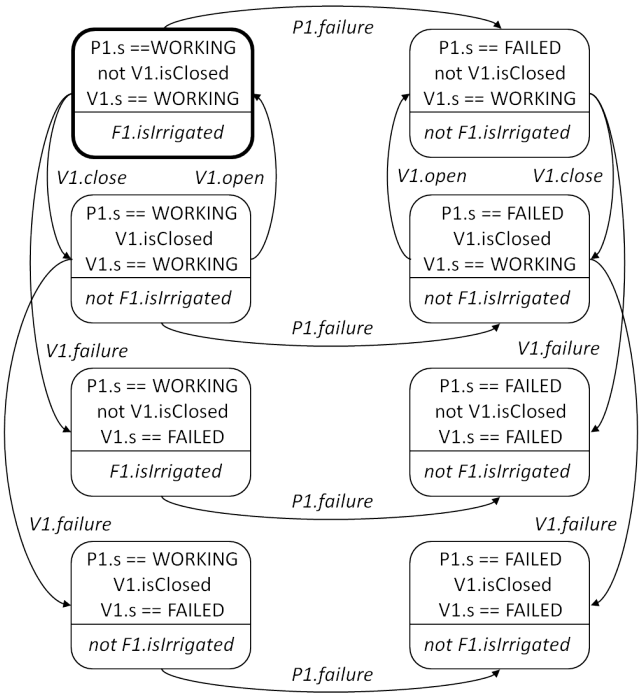


Figure 2.10: The Reachability graph of the system

- $\Sigma = \Sigma_1 \times \Sigma_2$, i.e. $\forall \sigma^1 \in \Sigma_1, \sigma^2 \in \Sigma_2$ the state $\sigma = \sigma^1 \circ \sigma^2$ belongs to Σ.
- If $(\sigma^1, e^1, \tau^1) \in \Theta_1$, then the set of transitions $\{(\sigma^1 \circ \sigma^2, e^1, \tau^1 \circ \sigma^2), \forall \sigma^2 \in \Sigma_2\}$ belongs to Θ.
- If $(\sigma^2, e^2, \tau^2) \in \Theta_2$, then the set of transitions $\{(\sigma^1 \circ \sigma^2, e^2, \sigma^1 \circ \tau^2), \forall \sigma^1 \in \Sigma_1\}$ belongs to Θ.

Definition 2.11 (A path in a reachability graph)**.** Let $\Gamma = (\Sigma, \Theta)$ be a reachability graph. A sequence of events $e_1, \ldots, e_n$ is a path in the reachability graph Γ if the transitions $(\sigma_1, e_1\sigma_2)$, $(\sigma_2, e_2, \sigma_3)$, $\ldots$, $(\sigma_n, e_n, \sigma_{n+1})$ belong to Θ.

Definition 2.12 (Extension of a reachability graph)**.** Let $G = \langle V = S \uplus F, E, T, A, \iota \rangle$ be a GTS and let $\Gamma = (\Sigma, \Theta)$ its reachability graph. Let $\langle V^*, A^*, \iota^* \rangle$ be an assertion such that

- $V \cap V^* = \emptyset$,
- A^* is an assertion built over the variables from $V \cup V^*$.

Then the reachability graph $\Gamma|_{<V^*, A^*, \iota^*>} = (\Sigma^*, \Theta)$, such that:
for each variable assignment $\sigma : V \to \mathcal{C} \in \Sigma$, the variable assignment $\sigma^* = \sigma|_{V \cup V^*}$ belongs to Σ^*:

1. $\forall v \in V \ \sigma^*(v) = \sigma(v)$
2. $\sigma^* = Propagate(A^*, \iota^*, \sigma^*)$,

is called the extension of the reachability graph Γ by $\langle V^*, A^*, \iota^* \rangle$

2.6 On the modeling of flow propagation

The ability to represent flows circulating through the system, as in Reliability Block Diagrams, is an important property for formalisms dedicated to Safety Analyses. In a Guarded Transition System

$$G = \langle V = S \uplus F, E, T, A, \iota \rangle,$$

flows circulating through the system are represented by means of flow variables F and an assertion A. Flow variables can be assigned only in the assertion (never in the action of a transition). An assertion is an instruction: it expresses dependencies between flow and state variables. After each transition firing, first the action of the transition is executed to update state variables S, then the assertion A is executed to calculate flow variables F according to the modified values of state variables.

Assertions are very convenient to represent remote interactions between components because they make it possible to create complex hierarchical models from simple component descriptions just by plugging these descriptions together. In other words, assertions are the glue between the components.

The successive versions of AltaRica differ mainly by the way assertions are set and calculated. Guarded Transition Systems introduce a fixpoint mechanism to calculate assertions. This fixpoint mechanism makes it possible to handle looped models, i.e. models with flow variables that depend instantaneously on one another (without the firing of any transition).

In this section, we formally define the notion of a looped model-based on the dependency relation of variables in the instruction. Then, we explain why at least a fixpoint mechanism is needed to handle looped models. Finally, we show that the fixpoint mechanism can be implemented in an efficient way, i.e. (more or less) in linear time with the respect of the size of the assertion, thanks to the ideas stemmed in theoretical computer science (Tarjan's algorithm to compute strongly connected components in graphs [103]) and in Artificial Intelligence (Unit Resolution [29]).

2.6.1 Dependency relation

Let us denote by $var(Exp)$ variables used in the expression Exp, and by $var(I)$ variables used in the instruction I. Basically

- if I is an empty instruction "*skip*", then $var(I) = \emptyset$;
- if I is in the form "$v := E$", then $var(I) = \{v\} \cup var(E)$;
- if I is in the form "if C then J", then $var(I) = var(C) \cup var(J)$;
- if I is in the form "$I_1; I_2$", then $var(I) = var(I_1) \cup var(I_2)$.

Definition 2.13 (Immediate dependency of variables)**.** Let v and w be two variables from V and let I be an instruction built over the variables from V. We say that v depends immediately on w in I if one of the following holds:

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

Definition 2.14 (Dependency of variables)**.** Let v and w be two variables from V and let I be an instruction built over the variables from V. We say that v depends on w in the instruction I, if there is a variable $u \in V$ such that v depends immediately on u in I and u depends on w in I.

The dependency relation of variables in an instruction I can be represented by a dependency graph.

Definition 2.15 (Dependency graph of an instruction)**.** Let us consider an instruction I built over variables from V. An oriented graph $G_D[I] = (V_D, E_D)$, where

- V_D is a set of vertices, each vertex is labeled by a variable from $var(I) \subseteq V$,

- E_D is a set of edges, such that if u and w are variables from $var(I)$, $v_u \in V_D$ and $v_w \in V_D$ are the corresponding vertices of the graph, then if u depends immediately on w in I then the edge $e_{uw} = (u, w)$ is in E_D,

is called a dependency graph of an instruction I.

Remarks:

- Let u and w be two variables from V, let I be an instruction built over variables from V and let G_D be a dependency graph of I. If u depends on w in I then there is a path from the vertex v_u to the vertex v_w in the graph G_D.
- Let I be an instruction built over variables from V and let $G_D[I]$ be its dependency graph. If there are two variables u, w from V such that u depends on w in I and w depends on u in I, then the graph $G_D[I]$ contains cycles. Moreover, the vertices v_u and v_w belong to the same strongly connected component.
- Let A be an assertion, i.e. an instruction built over the variables from $V = S \uplus F$ and let $G_D[A]$ be a dependency graph of A. If s is a state variable from $S \subseteq V$ and v_s the corresponding vertex, then the vertex v_s has no out-going edges.

Looped assertion

Definition 2.16 (Looped Instruction)**.** Let I be an instruction built over variables from V. We say that an instruction I is looped if it contains a variable that depends on itself in I:

$$\exists v \in var(I) | v \text{ depends on itself in } I$$

Example 2.10 (An irrigation system)**.** Consider the irrigation system from Example 2.1 and its GTS, given Figure 2.9. The dependency graph of the assertion is given Figure 2.11. Vertices labeled by state variables are marked in gray. Vertices labeled by flow variables are marked in white. Note that this graph contains cycles.

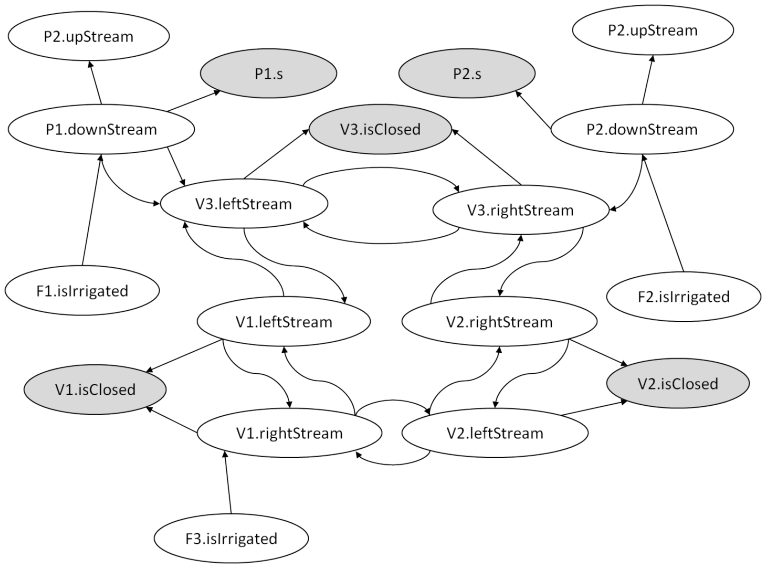


Figure 2.11: Dependency graph of the assertion of the Irrigation System

□

Data-Flow assertion

Definition 2.17 (Data-Flow Instruction)**.** Let I be an instruction built over the variables from V. I is said Data-Flow if no variable $w \in var(I)$ depends on itself in I.

Remark: Let I be an instruction built over the variables from V, and let $G_D[I]$ be its dependency graph. If I is a Data-Flow instruction, then $G_D[I]$ is a Directed Acyclic Graph (DAG). Topological sort of the vertices in the dependency graph $G_D[I]$ determines the order in which instructions from I should be evaluated.

In practice, this order is calculated during the compilation of the model and a reordering of blocks of instructions is performed. Thanks to the reordering of blocks of instructions, each assignment is executed only once. There is always a unique variable assignment satisfying a Data-Flow instruction.

Assume that n is the number of assignments in the assertion A which is Data-Flow. Then the complexity of the execution of the assertion A is $O(n)$, i.e. linear on the number of assignments.

Example 2.11 (A pumping system)**.** Consider a pumping system depicted Figure 2.12. It is made of a tank, two pumps connected in parallel and a reactor. The flow goes from the tank to the reactor through the pumps. Pumps may fail in operation and tank may become empty. The GTS representing such a pump is given Figure 2.3. The GTS representing a tank, a reactor and the hole pumping system is given Figure 2.13.

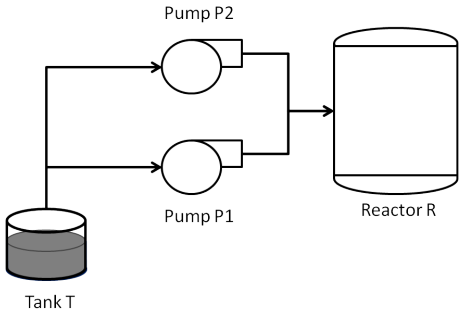


Figure 2.12: A pumping system

The assertion of the flattened GTS of the Pumping system is as follows:

```
T.outFlow := if not T.isEmpty then 100 else 0.0;
P1.outFlow := if P1.state==WORKING then P1.inFlow else 0.0;
P2.outFlow := if P2.state==WORKING then P2.inFlow else 0.0;
P1.inFlow := T.outFlow;
P2.inFlow := T.outFlow;
R.inFlow := P1.outFlow + P2.outFlow;
```

This assertion is a Data-Flow assertion. Its dependency graph is given Figure 2.14. Note that it is a Directed Acyclic Graph (DAG).

□

2.6.2 Handling looped models

Handling of looped systems requires to solve reachability problems, i.e. to determine whether a node is accessible from another one in a graph. In his paper on Guarded Transition Systems [90], Antoine Rauzy proposes to introduce the fixpoint mechanisms to calculate the assertion in order to handle looped systems. One iterates to accumulate reachable states until no more states can be added, i.e. a

```
class Tank
  Boolean isEmpty (init = FALSE);
  Real outFlow (reset = 0.0);
  event getEmpty;
transition
  getEmpty: not isEmpty -> isEmpty := TRUE;
assertion
  outflow := if not isEmpty then 100 else 0;
end
class Reactor
  Real inFlow (reset = 0.0);
end
class PumpingSystem
  Tank T;
  Pump P1, P2;
  Reactor R;
assertion
  P1.inFlow := T.outFlow;
  P2.inFlow := T.outFlow;
  R.inFlow := P1.outFlow + P2.outFlow;
end
```

Figure 2.13: GTS representing the Pumping system

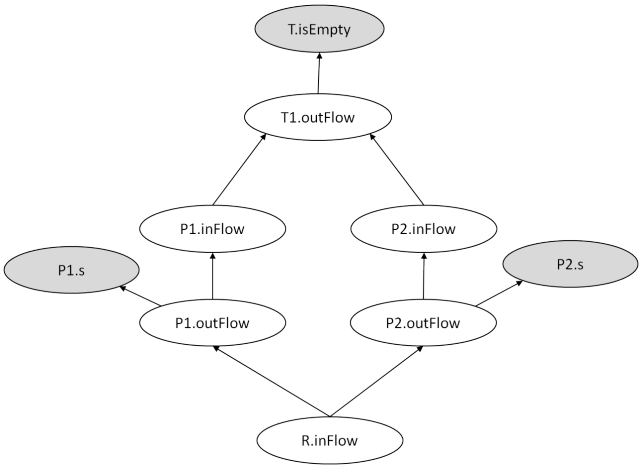


Figure 2.14: The Dependency graph of the assertion of the pumping system

fixpoint is reached. Not only the fixpoint semantics for the assertion provides an elegant solution to complex modeling problems, but it can be implemented in an efficient way, i.e. (more or less) in linear time with respect to the size of the assertion. That is what we show in the following subsection.

2.6.3 Algorithms to calculate assertions

The calculation of the fixpoint is performed as follows: first, one attempts to calculate (via the assertion) a value for each and every flow variable, assuming that these variables have no value at the beginning of the calculation; second, flow variables that remain free are assigned to their default value

and the assertion is used to check the consistency of the assignment.

In this section, we present different algorithms of fixpoint calculation.

Let A be an assertion, i.e. an instruction built over variables from V. Four types of instructions are defined for GTS: an empty instruction, an assignment, a conditional assignment and a parallel composition, also called a block of instructions.

It is possible to transform any block of instructions into a block containing only conditional assignments in the form $\langle v, C, E\rangle$, where v is a flow variable, C is a Boolean expression built over variables from V (i.e a condition), and E is an expression built over variables from V (i.e. a right hand side of an assignment). In case of a simple assignment, we consider that $C = \emptyset$. From now, the instruction of the form $\langle v, C, E\rangle$ is called an assignment.

In the following, we assume that the assertion A has been transformed into a block of assignments $A[i], i = 1..n$. Let n be the number of assignments in the assertion A.

Simple algorithm

Let D be a set of flow variables assigned in A: $D \subseteq F$. Let m be the number of variables in D, $m \leq n$. Let τ be a variable assignment obtained after the execution of the action of a transition. Let ι be the default variable assignment. Then the assertion A is calculated as follows.

```
ExecuteAssertion(A, ι, τ)
  D ← ∅
  GetAssignedVariables(A, D);
  ComputeValues(D, A, τ)
  ResetValues(D, ι, τ);
  VerifyValues(D, A, τ);
```

```
GetAssignedVariables(A, D)
  forall A[i] = ⟨v, C, E⟩, i = 1..n do
    D ← D ∪ {v}
  done
```

```
ComputeValues(D, A, τ)
  while D ≠ ∅ do
    int x = size(D);
    forall A[i] = ⟨v, C, E⟩, i = 1..n do
      ExecuteAssignment(⟨v, C, E⟩, D, τ);
    done
    if x==size(D) then break;
  done
```

```
ExecuteAssignment(⟨v, C, E⟩, D, τ)
  if C = ∅ or τ(C) = TRUE then
    if τ(E) ≠? then
      if τ(v) =? then
        1. τ(v) ← τ(E)
        2. D ← D \ {v}
      else
        if τ(v) ≠ τ(E) then ReportError; break;
```

```
ResetValues(D, ι, τ)
  ∀v ∈ D  τ(v) ← ι(v)
```

```
VerifyValues(D, A, τ)
  ∀v ∈ D, ∀⟨v, C, E⟩ ∈ A
    if C = ∅ or τ(C) = TRUE then
      if τ(v) ≠ τ(E) then ReportError; break;
```

In the worst case the complexity of the algorithm given above is $O(nm)$, where n is the number of assignments and m is the number of assigned flow variables, $n \geq m$.

Efficient algorithm

In this section, we propose a more efficient algorithm to calculate assertions. It relies on a unit propagation algorithm à la Dowling and Gallier [29].

Let D be a set of flow variables assigned in A: $D \subseteq F$. Let τ be a variable assignment obtained after the execution of the action of a transition. Let ι be the default variable assignment.

Before starting the calculation of the assertion A, let define a hash table H that associates with each variable $v \in var(A) \subseteq V$ a set of assignments $\{a_i = \langle w, C, E\rangle \in A | v \in var(C) \vee v \in var(E)\}$. Let U be a set of variables evaluated in τ. At the beginning, U contains state variables used in A. $U = \{s \in S | s \in var(A)\}$.

```
ExecuteAssertion(A, H, U, ι, τ)
  D ← ∅, W ← ∅
  GetAssignedVariables(A, D);
  ComputeValues(W, U, H, τ);
  D ← D \ W
  ResetValues(D, ι, τ);
  VerifyValues(D, A, τ);
```

```
ComputeValues(W, U, H, τ)
  while U ≠ ∅ do
    Let w ∈ U
    U ← U \ {w}
    forall H[w] = ⟨v, C, E⟩ do
      ExecuteAssignment(⟨v, C, E⟩, W, U, τ);
    done
  done
```

```
ExecuteAssignment(⟨v, C, E⟩, W, U, τ)
  if C = ∅ or τ(C) = TRUE then
    if τ(E) ≠? then
      if τ(v) =? then
        1. τ(v) ← τ(E)
        2. U ← U ∪ {v}
        3. W ← W ∪ {v}
      else
        if τ(v) ≠ τ(E) then ReportError; break;
```

Let k be the number of variables used in conditions C and expressions E of assignments of the assertion A. The complexity of the algorithm presented above is $O(k)$, i.e. linear on the number of variables used in the conditions and expressions of assignments of A.

Optimized assertion

The dependency graph of the assertion A, $G_D[A]$, is used to determine whether the assertion is a Data-Flow instruction. If $G_D[A]$ is a Directed Acyclic Graph (DAG), then A is a Data-Flow instruction. The topological sort of the graph determines the order of the execution of the assignments in A. Only one pass is needed to calculate all variables assigned in A.

If $G_D[A]$ has cycles, its graph of strongly connected components G_D^{SCC} should be considered. G_D^{SCC} enables to find groups of variables that should be calculated together. Indeed, variables that label vertices belonging to the same strongly connected component should be calculated together. The assertion A is decomposed into blocks of assignments $A = A_1; A_2; \ldots; A_r$, where r is the number of strongly connected components and A_j is a block of assignments $\langle v, C, E\rangle$ such that the variable v belongs to the strongly connected component j. The graph of strongly connected components G_D^{SCC} is a Directed Acyclic Graph (DAG). The topological sort of this graph determines the best order of execution of A_j.

The optimization algorithm works as follows:

1. BuildDependencyGraph(A, G_D);
2. BuildStronglyConnectedComponents(G_D, G_D^{SCC});
   Note that Tarjan's algorithm to compute strongly connected components of the graph [103] is linear on the number vertices in the graph (which is equal to the number of variables in $var(A)$).
3. DoTopologicalSort(G_D^{SCC}, A');

The result is the instruction A^* decomposed into blocks of assignments $A' = A_1; A_2; \ldots; A_r$ with blocks sorted in the best execution order.

A' is the optimized assertion. It is calculated as follows:

```
ComputeAssertion(A' = A_1; ...; A_r, ι, τ)
  forall A_j, j = 1..r do
    ExecuteAssertion(A_j, ι, τ);
  done
```

Thanks to optimization techniques based on Tarjan's algorithm to calculate strongly connected components of a graph, and thanks to the efficient algorithm to calculate fixpoints based on unit propagation à la Dowling and Gallier [29], it is possible to implement the calculation of fixpoints efficiently: the algorithm is linear on the size of the assertion (more precisely on the number of variables used in condition and right members of assignments).

2.6.4 Different approaches to interpret assertions

Different variants of AltaRica modeling language have been developed since its creation in nineties. Besides syntactic distinctions, different approaches have been proposed to interpret assertions (i.e. to propagate flows):

1. Solving constraints (Constraint automata) in the first version of AltaRica (also called AltaRica LaBRI);

2. Propagation (Mode automata) in AltaRica Data-Flow;

3. Fix point (Guarded Transition Systems) in AltaRica 3.0.

The goal of this section is to compare these different mechanisms of flow propagation and to discuss their advantages and drawbacks.

Constraint automata

The semantics of the first version of AltaRica modeling language (also called AltaRica LaBRI), developed at the Computer Science Laboratory of University of Bordeaux (LaBRI) in nineties, is expressed in terms of Constraint automata [80, 7]. Constraint automaton is a special kind of a finite state machine, where assertions are interpreted as constraints, i.e. after each transition firing, the flow variables are updated by solving constraints. It is a very powerful mechanism. Particularly, it is possible to represent bidirectional flows.

However, the experience has shown that the assessment algorithms are too resource consuming for industrial scale applications. For instance, it would be just impossible to call a constraint solver at each step of a stochastic simulation. Also, if the system of constraints has several solutions or does not have any solution at all, it can only be detected during the execution of the model, i.e. at run time.

Mode automata

To be able to handle industrial scale models, a new Data-Flow version of AltaRica has been proposed [88, 14]. The formal semantics of AltaRica Data-Flow is based on the notion of Mode automata [88] and can also be expressed in terms of Data-Flow Guarded Transition Systems [92].

A strong constraint is put on the assertion: it should be Data-Flow (see Definition 2.17). If the instruction is Data-Flow, it has a unique solution which is calculated in a very efficient way, only one pass is needed. The fact that an assertion does not satisfy the Data-Flow condition can be detected during the compilation of the model. Efficient assessment tools have been developed for AltaRica Data-Flow, such as compilers to Fault Trees and Markov chains, generator of critical sequences of events, model-checkers, stochastic and stepwise simulators.

However the Data-Flow condition put on the assertion prevents from modeling systems with instant loops, such as for example electrical or network systems. It is also difficult to represent acausal components, i.e. components for which the input and output flows are decided at run time (see example of the Irrigation system in Section 2.2).

Guarded Transition Systems

Guarded Transitions Systems (GTS) extend Mode automata: they remove the Data-Flow constraint for the assertion. Thereby GTS enable to represent systems with instant loops. The example of the Irrigation system, presented in Section 2.2, shows how acausal components can be represented. The direction of the flow is determined at run time for it depends on the global state of the system. Efficient fix point calculation algorithm is used in order to calculate flows: only one pass is needed for the propagation. All assessment algorithms developed for AltaRica Data-Flow can be extended for Guarded Transition Systems without significant loss of efficiency. Modeling errors (e.g. nonexistence of the fix point) are detected at run time.

Note that neither constraint resolution nor flow propagation make it possible to handle systems with instant loops. As seen in Section 2.6.2, at least a fixpoint mechanism to execute assertions is required.

Comparison of the three mechanisms of assertion calculation is summarized in Table 2.3.

Table 2.3: Comparison of flow propagation mechanisms

Variant	AltaRica LaBRI	AltaRica Data-Flow	AltaRica 3.0
Mathematical model	Constraint Automata	Mode Automata	GTS
Mechanism to update flows	Constraint solving	Propagation	Fix point
Algorithm efficiency	Low	High*	High**
Looped systems	No	No	Yes
Acausality	Yes	No	Yes
Error detection	Run time	Compilation	Run time

* - linear on the number of assignments, ** - linear on the number of variables used in the expressions of assignments

2.7 Timed/Stochastic Guarded Transition Systems

A probabilistic timed structure can be put on the top of Guarded Transition Systems.

2.7.1 Timed Guarded Transition Systems

Definition 2.18 (Timed Guarded Transition System)**.** A Timed Guarded Transition System is a tuple $\langle V, E, T, A, \iota, delay \rangle$, where

- $\langle V, E, T, A, \iota \rangle$ is a Guarded Transition System,
- $delay : E \to \mathbb{R}_+$ is a function, that associates to each event a non-negative real number.

The state of a Timed Guarded Transition System is a couple $(\sigma, d) \in \Sigma \times D$, where σ is a variable assignment as defined previously and $d : T \to \mathbb{R}_+$ is a delay assignment (i.e. a function that associated for each transition $tr \in T$ a delay $d(tr)$), $\Sigma = \prod_{v \in V} dom(v)$ is a set of all variable assignments, called states, and $D = {\mathbb{R}_+}^{\#T}$ is a set of delay assignments. If $d(tr) = 0$ then the transition tr should be fired.

The semantics of a Timed Guarded Transition System is defined in terms of Timed Transition Systems, including transitions labeled by the events from E and timed transitions labeled by real delays.

Definition 2.19 (Timed Transition System)**.** A Timed Transition System $\mathcal{S}$ is a quadruple $(S, s_0, \to, E)$, where S is a set of states, E is a set of events, $s_0 \in S$ is an initial state, $\to \subseteq S \times (E \cup \mathbb{R}_+) \times S$ is a transition relation.

Definition 2.20 (A run)**.** A run ρ of a Timed Transition System $\mathcal{S}$ is a sequence of transitions $\rho = s_0 \xrightarrow{l_0} s_1 \dots s_{n-1} \xrightarrow{l_{n-1}} s_n$, where $\forall\ 0 \leq i \leq n-1$, $s_i \xrightarrow{l_i} s_{i+1}$. A state $\sigma \in \Sigma$ is reachable if there is a run from σ_0 to σ.

Definition 2.21 (Semantics of Timed Guarded Transition Systems)**.** Given a Timed Guarded Transition System $TGTS = \langle V, E, T, A, \iota, delay \rangle$, the semantics of $TGTS$ is a Timed Transition System $\mathcal{S}_{TGTS} = (S, s_0, \to, E)$, such that:

- $S = \Sigma \times D$,

- $s_0 = (\sigma_0, d_0)$, where σ_0 is the initial state of the Guarded Transition System $\langle V, E, T, A, \iota \rangle$, calculated as follows:
$$\sigma_0 = Propagate(A, \iota, \iota),$$
and for each transition $tr = \langle e, G, P \rangle \in T, e \in E$, its initial delay is calculated as follows:
$$d_0(tr) = \begin{cases} delay(e), \text{if } \sigma_0(G) = TRUE, \\ +\infty, \text{otherwise.} \end{cases}$$

- $\rightarrow \subseteq S \times (E \cup \mathbb{R}_+) \times S$ describes two types of transitions:
  - transitions labeled by events: $(\sigma_i, d_i) \xrightarrow{e} (\sigma_{i+1}, d_{i+1})$ iff
    1. $\exists\ tr = \langle e, G, P \rangle \in T, e \in E$ such that $d_i(tr) = 0$,
    2. $\sigma_{i+1} = Fire(P, A, \iota, \sigma_i)$,
    3. for each transition $tr = \langle e, G, P \rangle \in T, e \in E$, its delay is calculated according to the new state σ_{i+1}
$$d_{i+1}(tr) = \begin{cases} delay(e), \text{if } \sigma_{i+1}(G) = TRUE, \\ +\infty, \text{otherwise.} \end{cases}$$
  - timed transitions: $(\sigma_i, d_i) \xrightarrow{\delta} (\sigma_i, d_{i+1})$ iff
    1. $\delta = \min_{tr \in T}(d_i(tr))$,
    2. for each transition $tr \in T$
$$d_{i+1}(tr) = \begin{cases} d_i(tr) - \delta, \text{if } \sigma_i(G) = TRUE, \\ +\infty, \text{otherwise.} \end{cases}$$

Informally speaking, the execution of a Timed Guarded Transition System is as follows: the system starts from the initial configuration or the initial state, where the initial system state σ_0 and the initial delays d_0 are calculated as defined previously, then two types of transitions are possible:

- If $\exists\ tr = \langle e, G, P \rangle \in T, e \in E$, such that its delay is equal to 0, then this transition is fired (new state and new delays are calculated).
- If there is no transition with a delay equal to 0, then the minimal delay δ is chosen and the time is advanced according to this value (the system state does not change but the delays decrease by δ).

A run of a Timed Transition System can be also seen as a timed word, i.e. a set of couples (e, t), where $e \in E$ is an event and $t \in \mathbb{R}_+$ is the date, when the event e occurred. A run can be also represented as a sequence $(\sigma_0, d_0, t_0) \xrightarrow{e_0} (\sigma_1, d_1, t_1) \ldots (\sigma_{n-1}, d_{n-1}, t_{n-1}) \xrightarrow{e_{n-1}} (\sigma_n, d_n, t_n)$, where $t_i \in \mathbb{R}_+$ represents the date when the event e_i occurs and the transition labeled by this event is fired. $t_0 = 0, t_{i+1} = t_i + \delta$. The step $(\sigma_i, d_i, t_i) \xrightarrow{e_i} (\sigma_{i+1}, d_{i+1}, t_{i+1})$ corresponds to the advancement of time and to the firing of the transition labeled by the event e.

2.7.2 Stochastic Guarded Transition Systems

The timed interpretation of GTS does not specify how delays are calculated. Therefore, it encompasses the case where delays are stochastic. Let define an oracle $o : \mathbb{N} \rightarrow [0; 1] \subset \mathbb{R}$, 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).

Definition 2.22 (Stochastic Guarded Transition System)**.** A Stochastic Guarded Transition System is a tuple $\langle V, E, T, A, \iota, delay, expectation\rangle$, where

- $\langle V, E, T, A, \iota\rangle$ is a GTS;
- delay is a function from events and oracles to non-negative real numbers $delay : E\times O \to \mathbb{T} \subseteq \mathbb{R}_+$;
- expectation is a function from events to positive real numbers $expectation : E \to \mathbb{R}_+$.

For the sake of simplicity, we made *delay* 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 calculation of delays $d_i(t), t \in T$ depends on an oracle o.
- When several transitions are scheduled to be fired at the same date, i.e. $\exists t_1, t_2, \ldots, t_k$ such that $d_n(t_1) = 0, d_n(t_2) = 0, \ldots, d_n(t_k) = 0$, one is picked at random by using the oracle and according to their expectations. The probability $p(e_k : G_k \to P_k)$ to fire the transition $e_k : G_k \to P_k$, is defined as follows.

$$p(e_k : G_k \to P_k) = \frac{expectation(e_k)}{\sum\limits_{t_i=\langle e_i,G_i,P_i\rangle : d_n(t_i)=0} expectation(e_i)} \tag{2.3}$$

Example 2.12 (A spare pump)**.** As an illustration, consider that in the pumping system from Example 2.11, there is a main pump and a spare one. The spare pump is normally in standby. It is attempted to start when it gets demanded, i.e. when the main pump fails. It is stopped as soon as the main pump is working again. The model for such a pump is pictured Figure 2.15.

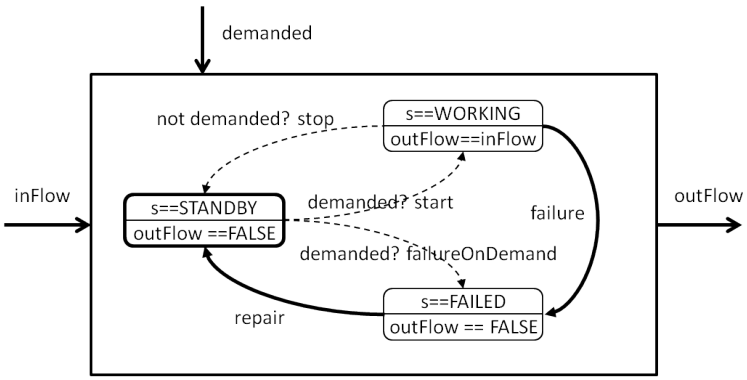


Figure 2.15: Stochastic GTS of a spare pump

On the GTS pictured Figure 2.15, transitions `stop`, `start` and `failureOnDemand` are deterministic and instantaneous: they are associated with a delay 0. They are represented by dashed lines. Transitions `failure` and `repair` are timed and stochastic. They obey typically exponential distributions with a failure rate λ and a repair rate μ. They are represented by plain lines. Finally, when the pump is attempted to start, it fails on demand with a probability γ, and is correctly started with a probability $1 - \gamma$.

The stochastic GTS representing a spare pump is given Figure 2.16. The component is initially

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

class SparePump
  StandbyComponentState s (init = STANDBY);
  Boolean demanded(reset = FALSE);
  Real outFlow, inFlow(reset = 0.0);
  parameter Real lambda = 1.0e-5;
  parameter Real mu = 0.01;
  parameter Real gamma = 0.02;
  event failure (delay = exponential(lambda));
  event repair (delay = exponential(mu));
  event failureOnDemand (delay = 0, expectation = gamma);
  event start (delay = 0, expectation = 1-gamma);
  event stop (delay = 0);
transition
  failure: s == WORKING -> s := FAILED;
  repair: s == FAILED -> s := STANDBY;
  start: s == STANDBY and demanded -> s := WORKING;
  failureOnDemand: s == STANDBY and demanded -> s := FAILED;
  stop: s == WORKING and not demanded -> s := STANDBY;
assertion
  outFlow := if s == WORKING then inFlow else 0.0;
end
```

Figure 2.16: Stochastic GTS code of a spare pump

in standby. When it is demanded, i.e. when the flow `demanded` is turned to `true`, the component is attempted to start. This operation takes no time (the delay is null) but may fail with a probability given by the parameter `gamma` or be successful with a probability (1-`gamma`). □

2.8 Comparison with classical formalisms for Safety Analyses

The prerequisites of a high level modeling language for Safety Analyses have been discussed in chapter 1. Within our original approach, they imply some expected properties for the underlying formalism. Here we review these properties and compare GTS with the classical formalisms (Booleans and States/Transitions) introduced in chapter 1.

Event based

The goal of Safety and Reliability studies is to determine the most probable failure scenarios, i.e. sequences of events leading the system from its nominal state to a failure state (an incident or an accident). So, the formalism should make it possible to describe systems in terms of states and transitions labeled with events. It should be possible to associate probability distributions to events. Events can be timed, or stochastic or immediate.

As mentioned in chapter 1, Boolean formalisms are naturally event-based. However, events are assumed to be completely independent, and only timed events (mainly failures) are considered.

Markov chains (MC) and Generalized Stochastic Petri nets (GSPN) are also event-based formalisms. Markov chains manipulate only exponentially distributed and immediate events. Generalized

Stochastic Petri nets deal with all the probability distributions.

Guarded Transition Systems (GTS) are also event based. Events can be immediate or timed. Different probability distributions can be associated with events in order to create Timed/Stochastic models (see section 2.7).

Implicit representation of a state space

For any reasonable size system, the number of reachable states and failure scenarios are just astronomical. It is impossible to represent all the states the system may reach. So, states must be given in an implicit way to avoid the exponential blow-up of the model and to allow approximations consisting in considering only the most probable states.

When modeling with Markov chains (MC), one needs to represent the state graph explicitly. However, Generalized Stochastic Petri nets (GSPN) give the state space in an implicit way.

As seen in section 2.4, GTS represent the state space of the system in an implicit way.

Composition

Systems are modeled recursively by assembling sub-systems and components. The formalism should be compositional in order to make it possible to describe systems as hierarchies of components and to assemble models of components "in a Lego way".

Boolean formalisms are naturally hierarchical (see e.g. Figure 1.4 for Reliability Block Diagrams (RBD)). However, classical States/Transitions formalisms, Markov chains and Generalized Stochastic Petri nets, do not have this property.

As seen in section 2.4, GTS can be assembled into hierarchies of reusable components by means of three operations: the free product, the connection and the synchronization.

Remote interactions

Fault Trees or Reliability Block Diagrams make it possible to describe instant remote interactions between components of the system under study. It should be possible to describe easily remote interactions between components, i.e. flows of matter or information circulating through the system.

Markov chains and Generalized Stochastic Petri nets do not allow to easily represent remote interactions.

Thanks to flow variables and assertions (see section 2.6), GTS can easily represent the propagation of flows. Synchronizations can also be used to represent remote interactions between components.

Efficiency of the assessment algorithms

Efficient assessment algorithms should be available to assess models.

As seen in chapter 1, very efficient assessment algorithms are available for Boolean formalisms. There are tools, able to deal with Markov chains having up to one million states. Generalized Stochastic Petri nets can be efficiently assessed by a stochastic simulation.

All the assessment algorithms, developed for AltaRica Data-Flow (see section 1.4), can be extended to GTS without loosing their efficiency:

- GTS can be compiled into Fault Trees and critical sequences of events. It is the subject of the fourth chapter of this thesis.
- GTS can be compiled into Markov chains [21].
- GTS can be assessed by a stochastic simulation [9], etc.

GTS are a States/Transitions formalism that generalizes Reliability Block Diagrams (RBD) and Markov chains (MC). GTS have all the good properties of Boolean formalisms. As seen in section 2.6, thanks to the fixpoint mechanism to calculate assertions, GTS make it possible to represent systems with acausal components and to handle looped systems, such as electrical systems or communication networks. Comparison between GTS and classical formalisms for Safety Analyses is summarized in Table 2.4.

Table 2.4: Comparison of the formalisms for Safety Analysis

	FT	RBD	MC	GSPN	GTS
Event-centric	Yes	Yes	Yes	Yes	Yes
States/Transitions	No	No	Yes	Yes	Yes
Implicit representation of state space	Yes	Yes	No	Yes	Yes
Hierarchical representation	Yes	Yes	No	No	Yes
Remote interactions	Yes	Yes	No	No	Yes
Efficient assessment tools	Yes	Yes	Yes	Yes	Yes

Summary

In this chapter we presented Guarded Transition Systems (GTS) – a States/Transitions formalism dedicated to Safety Analyses. GTS have all the good modeling properties, as summarized in Table 2.4. They have a versatile synchronization mechanism, which enables to easily represent common cause failures, shared resources, etc. In addition, they make it possible to represent acausal components and to handle looped systems.

GTS generalize classical formalisms for Safety Analysis, such as Markov chains and Reliability Block Diagrams. They can be seen as a pivot formalism for Safety studies. Other Safety models can be compiled into Guarded Transition Systems in order to take advantage from the assessment tools.

GTS are the underlying mathematical formalism of AltaRica 3.0. The next chapter is dedicated to the structural constructs of AltaRica 3.0.

Chapter 3

System Structure Modeling Language (S2ML)

Models of large, multi-scale systems are necessarily large and complex. Therefore, they need to be well structured to ensure their maintainability through the life cycle of systems as well as the capitalization of knowledge from project to project. Thus, high level modeling languages should not only embed the suitable mathematical concepts, but also provide powerful constructs to structure models.

This chapter introduces structural constructs of AltaRica 3.0. To a large extent, these structural constructs are independent from the underlying mathematical formalism of AltaRica 3.0. They are assembled into System Structure Modeling Language (S2ML).

Many system engineering modeling languages (e.g. Modelica [38]) relies on the object-oriented paradigm to structure models. AltaRica 3.0 enriches it with some prototype-oriented constructs (for a detailed presentation of prototype-oriented languages see e.g. [73]) which correspond better to the abstraction level of Safety and Reliability models.

Any hierarchical AltaRica 3.0 model can be transformed into an equivalent flat model (i.e. a model without any hierarchy). This operation is called *flattening*.

This chapter is organized as follows. Section 3.1 presents the motivations for integrating new structural constructs into AltaRica 3.0. Section 3.2 summarizes some differences between the object-oriented and the prototype-oriented paradigms. Section 3.3 describes the structural constructs of the language. Section 3.4 is dedicated to structural operations which make it possible to create hierarchical models. Section 3.5 defines flattening rules. Section 3.6 discusses several important aspects of modeling and lists some still open questions.

3.1 Motivations

Different paradigms to structure models are used in programming and high level modeling languages. In functional languages the components are represented by functions with inputs and outputs; outputs of functions depend only on their inputs and not on the program state. This principle is implemented in Lucid Synchrone [24] – a modeling language for the implementation of reactive systems.

In object-oriented programming languages, components of systems are represented by classes. Classes are stored in libraries and can be reused by instantiation or by extension [3]. In order to create hierarchical models, a class must embed instances of other classes. Modelica [37], a high level modeling language dedicated to dynamic behavior modeling and simulation of systems, and FIGARO [15], a modeling language for dependability studies developed by EDF R&D, adopt this paradigm to structure models. Previous versions of AltaRica modeling language (AltaRica LaBRI [80, 7] and AltaRica Data-Flow [88, 14]) use the paradigm of structured programming to organize models into hierarchies. Components of systems are represented by classes, called nodes, that can be stored in libraries. Nodes

can be reused by instantiation. A node embeds instances of other nodes in order to create hierarchical models.

However, experience shows that the object-oriented paradigm is not really adapted to structure safety models. First, object-orientation implies that each hierarchical level of the model should be an instance of a class. But in Safety Analyses, due to the abstraction level of models, most of the instances are unique. Consequently the creation of a class stored in a library for each hierarchical level is not really justified. To illustrate this point let's consider the following power supply system.

Example 3.1 (A Power Supply system)**.** Figure 3.1 depicts a power supply system, adopted from [16]. The system is composed of two redundant supply systems:

- a *Primary Power Supply* system, and
- a *Backup Power Supply* system,

to avoid the total system failure. The *Primary Power Supply* system is composed of two redundant supply lines. In the normal mode, the energy is provided via the first supply line: from the grid *GR* via the first upper circuit breaker (*CBU1*), the first transformer (*TR1*) and the first down circuit breaker (*CBD1*). In case of a failure, the circuit breakers *CBU2* and *CBD2* are attempted to close and the energy is provided via the second supply line. If the *Primary Power Supply* system is failed (when both lines fail to provide the power to the *Busbar*), a *Backup Power Supply* system is used. The *Backup Power Supply* system is made of a diesel generator *DG* and a circuit breaker *CB3*. All components of the *Power Supply* system can fail. An undesirable event occurs if the system fails to provide the power to the *Busbar*.

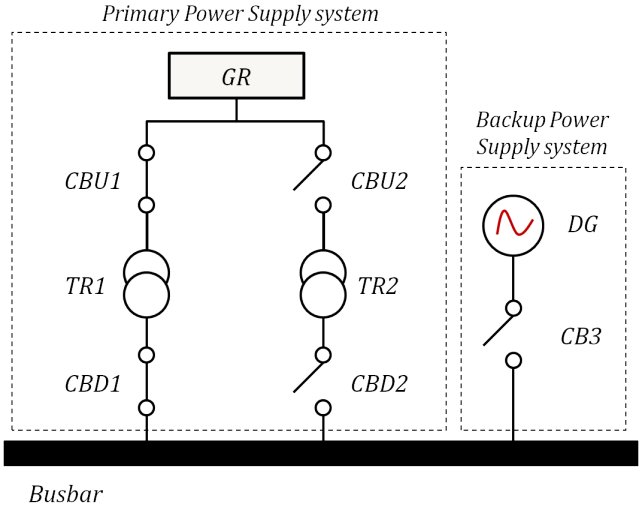


Figure 3.1: Power Supply System

□

According to the object-oriented paradigm,

- Each type of component is represented by a class (e.g. classes "CircuitBreaker", "Transformer", "Grid" and "DieselGenerator" model respectively circuit breakers, transformers, grids and diesel generators).
- *CBU1*, *CBD1*, *CBU2*, *CBD2* and *CB3* are instances of the class "CircuitBreaker", *TR1* and *TR2* are instances of the class "Transformer", *GR* is an instance of the class "Grid" and *DG* is an instance of the class "DieselGenerator".

- To create the hierarchical levels, two additional classes should be created: "PrimaryPowerSupplySystem" and "BackupPowerSupplySystem". The former embeds the instances *CBU1*, *CBD1*, *CBU2*, *CBD2*, *TR1*, *TR2* and *GR*. The latter embeds the instances *CB3* and *DG*.
- The model of the whole system embeds an instance of the class "PrimaryPowerSupplySystem" and an instance of the class "BackupPowerSupplySystem".

The model is given in Figure 3.2.

```
class PrimaryPowerSupplySystem
  Grid GR;
  CircuitBreaker CBU1, CBU2, CBD1, CBD2;
  Transformer TR1, TR2;
end

class BackupPowerSupplySystem
  DieselGenerator DG;
  CircuitBreaker CB3;
end

class PowerSupplySystem
  PrimaryPowerSupplySystem primarySystem;
  BackupPowerSupplySystem backupSystem;
end
```

Figure 3.2: Power Supply System model according to the object-oriented paradigm

Unlike the classes "CircuitBreaker", "Transformer", "Grid" and "DieselGenerator", created to represent individual components and having several instances (classes "Grid" and "DieselGenerator" are instantiated only once but we can imagine other power supply systems, where these models can be reused), the classes "PrimaryPowerSupplySystem" and "BackupPowerSupplySystem", used to create the hierarchical levels, have unique instances.

In addition, according to the object-oriented paradigm each class instance (i.e. each component, each sub-system) should be contained only in one class instance.

For example, the *Primary Power Supply* system can be broken down into two sub-systems: *Line1* and *Line2* (see Figure 3.3). *Line1* embeds components *GR*, *CBU1*, *TR1* and *CBD1*. *Line2* embeds components *GR*, *CBU2*, *TR2* and *CBD2*. The grid *GR* is shared by the sub-systems *Line1* and *Line2*. The instance "GR" cannot belong to the instances "Line1" and "Line2" at the same time, so, this decomposition is not in agreement with the object-oriented paradigm.

In other words, according to the object-oriented paradigm, models can only be structured in trees, i.e. each node of the tree can have only one parent. If we consider only the physical architecture of the system, the object-oriented paradigm to structure models can be sufficient. But if we also want to describe the functional architecture, there are difficult problems to overcome.

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 the power to the *Busbar*" in our example. The basic events of the Fault Tree, however, are almost always failures of physical components. A natural consequence of this association is that the failure of one component can impact different functions. In fact, Fault Trees are not structured in trees (contrary to what their name suggests). They are Directed Acyclic Graphs, because in a Fault Tree a basic event or a gate can have multiple parents. See, for example, Figure 3.4 representing the Fault Tree corresponding to the *Primary Power Supply* system, where the basic event "GR failed" has two parents: gates "Line1

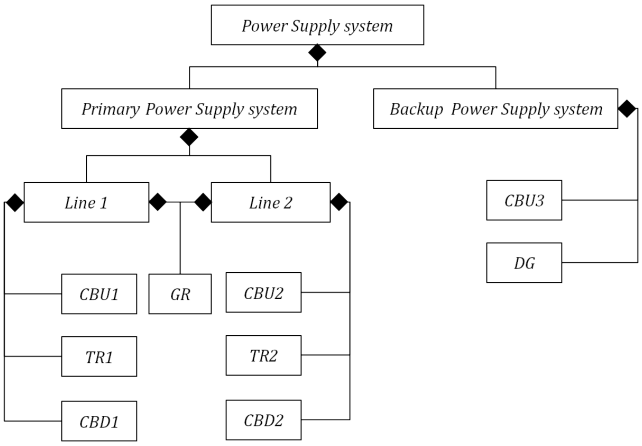


Figure 3.3: Power Supply System: break down structure

failed " and "Line2 failed". As a conclusion, the object-oriented paradigm is not adapted to represent Fault Trees – the most commonly used formalism in Safety Analysis.

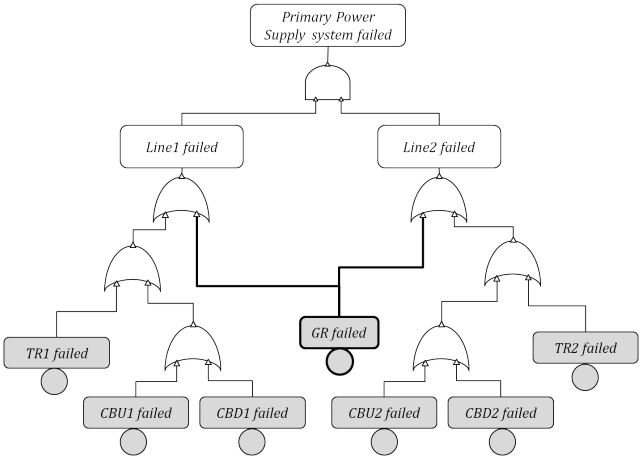


Figure 3.4: Primary Power Supply System: Fault Tree view

In the area of system architecture, this issue was discussed through the concept of functional chain (see e.g. [105]). The idea is that the system architecture (functional and physical) is never given globally, but via the functional chains or via the components involved in the implementation of the tasks of the system. These functional chains may not only share components, but they do not generally overlap the physical decomposition of the system.

The discussed problems do not appear in modeling languages for physical simulation, such as Matlab / Simulink or Modelica, precisely because they are focused on the physical architecture of the system under study and do not consider its functional aspects.

3.2 Object-oriented paradigm vs. prototype-oriented paradigm

Choosing a paradigm to structure models is closely related to the method used to construct models. Indeed, at least implicitly, every modeling formalism relies on and defines such a method. We can classify the methods in two large families:

"Top-down" approach: The model is built via a top-down system analysis. Amongst the formalisms supporting the "top-down" approach we can typically found SysML [36], BPMN [108], StateCharts [48], but also Fault Trees, Reliability Block Diagrams and Generalized Stochastic Petri nets [65].

"Bottom-up" approach: The model is constructed by assembling components (possibly grouped into libraries of "on-the-shelf" components). Amongst the formalisms supporting the "bottom-up" approach we can found Modelica [37] and Lustre [47].

This distinction goes beyond the issues of modeling. For example, the same idea can be found in Hatchuel C-K theory [49, 50] – a unified design theory. The central idea of this theory is the distinction between two spaces:

The K-space: the space of knowledge, that represents the stabilized knowledge, e.g. libraries of reusable components.

The C-space: the space of concepts, that regroups knowledge under development, i.e. the "new ideas".

The process of design is defined as a double expansion of the C and K spaces through the iterative exchange between them: the existing knowledge becomes the initial concept, then the concept is explored, new knowledge is added to the K-space, the knowledge from the K-space is then reused in the C-space to create new concepts, which in turn become new knowledge and are added to the K-space, etc.

In the domain of model design, the K-space can be seen as the space of stabilized knowledge, i.e libraries of "on-the-shelf" components, whereas the C-space is the "sandbox", the space where the model is created. The components from the libraries are reused in order to create new models (prototypes). In the "sandbox" some components are unique, some others come from the libraries. When a model is finalized, it is added to the library and can be reused for further modeling. This process is illustrated in Figure 3.5.

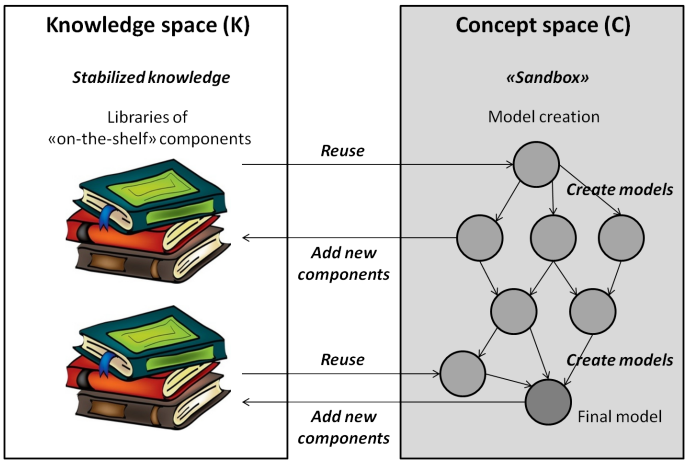


Figure 3.5: C-K theory applied to model design

In the domain of system engineering, the distinction between methods may be related to the level of abstraction at which one considers the system:

- the "top-down" methods are preferred when the system is analyzed globally,
- while the "bottom-up" methods are preferred at a lower level of abstraction.

These methods correspond to different mechanisms of models structuring. The "bottom-up" methods correspond to the object-oriented paradigm, while the "top-down" methods – to the prototype-oriented paradigm.

As mentioned before, in object-oriented languages, models are reused by instantiation of already defined classes [3]. In prototype-based programming languages [73] models are reused by cloning and modifying a model designed for a previous project. Examples of prototype-based programming languages are the programming language SELF [73] or, more recently, Javascript.

Differences between object-oriented and prototype-oriented paradigms to structure models are summarized in Table 3.1.

Paradigm	**Object-oriented**	**Prototype-oriented**
Concepts	class & object	prototype
Reuse principle	instantiation and inheritance	cloning and modifying
Corresponding system analysis method	"Bottom-up"	"Top-down"
Abstraction level	Low level analysis	Global analysis
Corresponding type of models reuse	Libraries of classes	Modeling patterns
C-K theory	K-space stabilized knowledge	C-space knowledge under development
Examples	Modelica, Figaro	SELF, Javascript

Table 3.1: Object-oriented paradigm vs. prototype-oriented paradigm

Safety models (and therefore AltaRica models) are ambivalent. Because they consider systems with a high level of abstraction, they naturally emerge from the prototype-oriented paradigm; but because they reuse components (e.g. to take into account the redundancy), they also emerge from the object-oriented paradigm.

3.3 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*.

3.3.1 *Blocks*

Definition 3.1 (Block)**.** *Block* is a structural construct that represents a prototype, i.e. a component having a unique occurrence in the model.

Let's consider the *Primary Power Supply system* from the Example 3.1. Figure 3.6 shows the structural part of AltaRica 3.0 code, which represents its hierarchical structure, depicted in Figure 3.3.

This (partial) model contains a hierarchy of nested blocks. Each block is unique. The behaviour of each component should be defined individually.

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*.

```
block PrimaryPowerSupplySystem
  block Grid ...end
  block Line1
    block CBU1 ...end
    block TR1 ...end
    block CBD1 ...end
  end
  block Line2
    block CBU2 ...end
    block TR2 ...end
    block CBD2 ...end
  end
end
```

Figure 3.6: Illustration of *block* usage

However, if two transformers are identical, which is probably the case, duplicate models is both tedious and error prone (copy and paste, update, ...). The idea is to define a generic component "Transformer" that can be then instantiated several times in the model. The definition of such a generic component is achieved via a *class*.

3.3.2 *Classes*

Definition 3.2 (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.

The behavior of the transformer can be defined by a Guarded Transition System (see chapter 2), depicted in Figure 3.7. The corresponding AltaRica 3.0 code is given Figure 3.8.

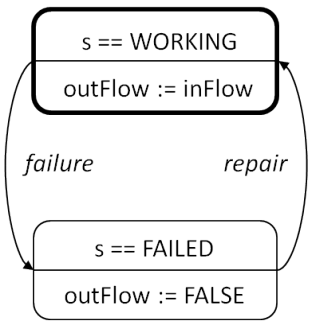


Figure 3.7: The behavior of the transformer

Instead of using *blocks* to define the behavior of individual components, *classes* are used to represent the transformer, the circuit breakers and the grid. They are then instantiated in the model as shown by the code given Figure 3.9.

Definition 3.3 (Object)**.** An instance of a class is called an *object*.

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.

```
domain RepairableComponentState { WORKING, FAILED }

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

Figure 3.8: The AltaRica 3.0 code of the transformer

```
block PrimaryPowerSupplySystem
  Grid GR;
  block Line1
    CircuitBreaker CBU1, CBD1;
    Transformer TR1;
  end
  block Line2
    CircuitBreaker CBU2, CBD2;
    Transformer TR2;
  end
end
```

Figure 3.9: Illustration of *class* usage

3.3.3 Using *Classes* or *Blocks*?

Some modeling languages like Modelica and the previous versions of AltaRica implement only the concept of *class*. Others, such as SysML, only have the notion of *block*. AltaRica 3.0 introduces both.

In safety models a lot of components are unique. In our example the "PrimaryPowerSupply" system, the "BackupPowerSupply" system, and also the lines inside the "PrimaryPowerSupply" system have unique occurrences in the model. In fact, they are more organizational or functional entities than physical components. Even if formally it is possible to represent them by *classes*, it is however better to distinguish them from generic components. Thus, it is preferred to model them by *blocks*.

AltaRica 3.0 makes a clear distinction between:

- the stabilized knowledge which is incorporated into libraries of "on-the-shelf" modeling components, for which *classes* are used; and
- the "sandbox" in which the analyst is designing his model of the system under study. In the sandbox, many components are unique; some others are instances of reusable components.

Declaring a *class* is in some sens creating another "sandbox". Amongst other consequences, this means that it is not possible to refer in a class to an object which is declared outside of the class, neither to declare a class into a class or a block. A class may of course contain blocks and instances

of other classes up to the condition that this introduces no circular definitions (recursive data types are not allowed in AltaRica 3.0).

Blocks may contain other blocks and objects. They can also aggregate the objects and blocks defined outside of them.

To summarize, a hierarchical model is a set of nested components. Components can be of two sorts:

- *Blocks* (also called prototypes), i.e. components that have a unique occurrence in the model.
- *Objects* (also called instances of classes), i.e. components that are created by cloning a generic component described separately.

All components are identified by a unique name.

Classes and *blocks* are declared in a similar way. Figure 3.10 presents grammar rules to declare *classes* and *blocks*.

```
Model ::= (Declaration)*
Declaration ::= ClassDeclaration | BlockDeclaration
ClassDeclaration ::=
      "class" IDENTIFIER
        ExtendsClause* DeclarationClause* BehaviorClause*
      "end"
BlockDeclaration ::=
      "block" IDENTIFIER
        ExtendsClause* (EmbedsClause | DeclarationClause)* BehaviorClause*
      "end"
ExtendsClause ::= "extends" Path Attributes? ";"
EmbedsClause ::= "embeds" Path "as" IDENTIFIER ";"
DeclarationClause ::= ObjectDeclaration | BlockDeclaration
ObjectDeclaration ::= Type IDENTIFIER ("," IDENTIFIER)* Attributes? ";"
Type ::= "Boolean" | "Integer" | "Real" | "Symbol" | Path
Attributes ::= "(" Attribute ("," Attribute)* ")"
Attribute ::= IDENTIFIER "=" Expression
Path ::= IDENTIFIER ("." IDENTIFIER)*
```

Figure 3.10: Declaration of structural constructs

3.4 Structural operations

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

3.4.1 Composition (*declaration* clause)

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

In the example given Figure 3.9, the *block* "Line1" contains an instance of the *class* "Transformer" named "TR1". We say that the *block* "Line1" is composed of the instance "TR1". Similarly, the *block* "PrimaryPowerSupplySystem" is composed of the *blocks* "Line1" and "Line2".

A *block* and also a *class* can be composed of *blocks* and *objects* (see Figure 3.13) 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.

When a *class* A (or a *block* B) is composed of an instance of a *class* C, named c, then the components 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. Thus, it is possible to refer the state of the transformer "TR1" in the *block* "PrimaryPowerSupplySystem" via the identifier "Line1.TR1.s". The prefixes "Line1" and "TR1" result from the compositions.

3.4.2 Inheritance (*extends* clause)

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

In the Example 3.1, a transformer and a grid (and also probably circuit breakers and a diesel generator) are repairable components. So, the model of the transformer must extend the model of the repairable component, and not be composed of it.

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 3.11. The *class* "Transformer" extends the *class* "RepairableComponent" and adds to it the flow variables representing the input and the output streams and the corresponding assertion. This definition of the *class* "Transformer" is equivalent to the one given Figure 3.8.

```
domain RepairableComponentState { WORKING, FAILED }

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

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

Figure 3.11: *extends* clause

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

```
block PrimaryPowerSupplySystem
  Grid GR;
  block Line1
    embeds GR as GRID;
    CircuitBreaker CBU1, CBD1;
    Transformer TR1;
  end
  block Line2
    embeds GR;
    CircuitBreaker CBU2, CBD2;
    Transformer TR2;
  end
end
```

Figure 3.12: *embeds* clause

the components of C in the *class* A (or in the *block* B) directly by their names. For example, in the *class* "Transformer" that extends the *class* "RepairableComponent" (see Figure 3.11), the state variable "s" of the latter is referenced in the former by its name (without any prefix).

3.4.3 Aggregation (*embeds* 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 *class instance* is contained in only one *block* or in only one *class instance*. Thus, the component *Grid* from the Example 3.1 cannot be contained in the block "Line1" and in the block "Line2" 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 the power to the Busbar" in the Example 3.1. 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 *blocks* and *objects* 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 3.12. The instance "GR" of the *class* "Grid" is shared between the *blocks* "Line1" and "Line2". Inside the *block* "Line1" the embedded *object* "GR" is given an alias "GRID". Thus it can be referenced inside of the *block* "Line1" either as "GR" or as "GRID".

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* "PrimaryPowerSupplySystem") or in the *class* in which it is declared. A *block* declared inside a *class* cannot be referenced outside of this *class*.

The comparison between *classes* and *blocks* is summarized in Table 3.2. Relations between *classes*, *objects* and *blocks* are represented in the diagram Figure 3.13.

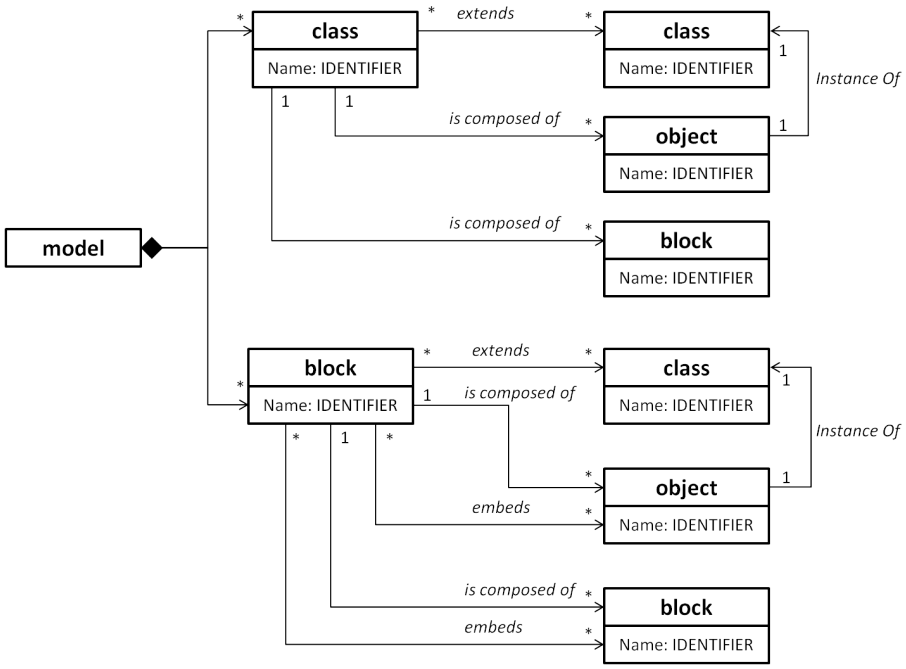


Figure 3.13: Relations between *classes*, *objects* and *blocks*

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"
C-K theory	K-space	C-space
Composition	Contains *objects & blocks*	Contains *objects & blocks*
Inheritance	Extends *classes*	Extends *classes*
Aggregation	–	Embeds *objects & blocks*

Table 3.2: Classes vs. Blocks

3.4.4 Relations between components

Besides the "vertical" relations between components (composition, inheritance), there are also their "horizontal" links, i.e. the means by which they interact. AltaRica 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.

Assertions

In the model of the *Primary Power Supply* system given Figure 3.12 all components are independent. To represent the propagation of power from the *grid* to the *Busbar*, we need to connect some flow variables together. To do that, several assertions are added to the previous model (see Figure 3.14).

```
block PrimaryPowerSupplySystem
  Boolean outFlow(reset = false);
  Grid GR;
  block Line1
    Boolean outFlow(reset = false);
    embeds GR as GRID;
    CircuitBreaker CBU1, CBD1;
    Transformer TR1;
  assertion
    CBU1.inFlow := GRID.outFlow;
    TR1.inFlow := CBU1.outFlow;
    CBD1.inFlow := TR1.outFlow;
    outFlow := CBD1.outFlow;
  end
  block Line2
    Boolean outFlow(reset = false);
    embeds GR;
    CircuitBreaker CBU2, CBD2;
    Transformer TR2;
  assertion
    CBU2.inFlow := GR.outFlow;
    TR2.inFlow := CBU2.outFlow;
    CBD2.inFlow := TR2.outFlow;
    outFlow := CBD2.outFlow;
  end
assertion
  outFlow := Line1.outFlow or Line2.outFlow;
end
```

Figure 3.14: AltaRica 3.0 model of the *Primary Power Supply* system: 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". 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 3.7 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 "Transformer" in Figure 3.11 are flow variables. In the model given Figure 3.14 the input flow of "CBU1" is connected to output flow of the grid "GRID", the input flow of the transformer is connected to the output flow of "CBU1", etc.

Assertions are presented in details in section 2.6.

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 transformers we need to add the new event "CCFTransformers" and the corresponding transition in the block "PrimaryPowerSupplySystem" (its AltaRica code is given Figure 3.14).

The new model is given Figure 3.15. The transition "CCFTransformers" is fireable if at least one

```
block PrimaryPowerSupplySystem
  ...
  event CCFTransformers;
  ...
transition
  CCFTransformers : ?Line1.TR1.failure & ?Line2.TR2.failure ;
  ...
end
```

Figure 3.15: AltaRica 3.0 model of the *Primary Power Supply* system: synchronizations

of the synchronized transitions is fireable.

Synchronizations are described in details in section 2.4.

3.5 Flattening

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.

3.5.1 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 objects (instances of classes) and atomic elements, i.e. 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 ($b \in B_A$) is flattened into a block b_{flat}.
   - For each block b, the block b_{flat} is added to C_{flat}.
4. Then, declared objects from D are flattened:
   - Each class Q such that C is composed of an instance q of Q ($q \in D$) 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 objects (variables, events, parameters, observers and their references in expressions and synchronizations) are prefixed with the name of the object followed by a dot.
5. Next, simple variable declarations, event declarations, parameter declarations and observer declarations of C are copied to C_{flat}. By simple 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}.

This algorithm is represented Figure 3.16, where

- the function `nameOf(`o`)` returns the name of its argument o,
- the function `Copy(`X`, `Y`)` copies elements of X into Y, and
- the function `CopyWithPrefix(prefix, `X`, `Y`)` copies elements of X into Y, during the copy all named elements (i.e. variables, observers, parameters, events and their references in expressions and synchronizations) are prefixed by `prefix` followed by a dot.

```
FlattenClass(C, C_flat)
  C_flat ← {∅}
  forall K ∈ C_E
    FlattenClass(K, K_flat);
    Copy(K_flat, C_flat);
  done
  forall b ∈ B_A
    FlattenBlock(b, b_flat);
    Copy(b_flat, C_flat);
  done
  forall d ∈ D
    if d == instanceOf(Q) then
      FlattenClass(Q, Q_flat);
      CopyWithPrefix(nameOf(d), Q_flat, C_flat);
    else
      Copy(d, C_flat);
  done
  Copy(T, C_flat);
  Copy(A, C_flat);
```

Figure 3.16: *Class* flattening

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 blocks and objects B_E,
- a set of declared elements D, including objects (instances of classes) and atomic elements, i.e. simple 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 ($b \in B_A$) is flattened into a block b_{flat}.
   - For each block b, b_{flat} is added to B_{flat}.
4. Then, declared objects of D are flattened:
   - Each class Q such that B is composed of an instance q of Q ($q \in D$) 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 objects (variables, events, parameters, observers and their references in expressions and synchronizations) are prefixed with the name of the object followed by a dot.

5. Next, simple 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 blocks and objects.

6. Finally, the behaviour clause is flattened:

   - Transitions of B are copied to B_{flat}.
   - Assertions of B are copied to B_{flat}.
   - During the copy references to variables in expressions are prefixed by the name of the block followed by a dot. References to variables and events belonging to embedded blocks and objects are not prefixed. Aliases of embedded blocks and objects are replaced by their paths.

The algorithm is depicted Figure 3.17, where the function `CopyWithPrefixExceptEmbeds(prefix,` X`,` Y`,` Z`)` copies elements of X into Z, during the copy all named objects (variables, parameters, events and observers and their references in expressions and synchronizations) are prefixed by `prefix` followed by a dot, except for those that belong to elements from Y(their aliases are also replaced by the paths of the elements from Y).

```
FlattenBlock(B, B_flat)
  B_flat ← {∅}
  forall K ∈ C_E
    FlattenClass(K, K_flat);
    Copy(K_flat, B_flat);
  done
  forall b ∈ B_A
    FlattenBlock(b, b_flat);
    Copy(b_flat, B_flat);
  done
  forall d ∈ D
    if d == instanceOf(Q) then
      FlattenClass(Q, Q_flat);
      CopyWithPrefix(nameOf(d), Q_flat, B_flat);
    else
      CopyWithPrefixExceptEmbeds(nameOf(B), d, B_E, B_flat);
  done
  CopyWithPrefixExceptEmbeds(nameOf(B), T, B_E, B_flat);
  CopyWithPrefixExceptEmbeds(nameOf(B), A, B_E, B_flat);
```

Figure 3.17: *Block* flattening

Example 3.2 (A Primary Power Supply system)**.** Consider the model of the *Primary Power Supply* given Figure 3.14. Moreover suppose that classes "CircuitBreaker" and "Grid" extend the class

"RepairableComponent", in the same way as the class "Transformer"(see Figure 3.11). Figure 3.18 illustrates how the algorithm of hierarchy flattening works; it presents the flattened *Primary Power Supply* system corresponding to the model given Figure 3.14.

□

3.5.2 Flattening of the synchronizations

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

$$e: \ !a_1 \ \& \ \ldots \ \& \ !a_m \ \& \ ?b_1 \ \& \ \ldots \ \& \ ?b_n \ \& \ L_1 \rightarrow R_1 \ \& \ \ldots \ \& \ L_r \rightarrow R_r,$$

$$m \geq 0, n \geq 0, r \geq 0,$$

where

1. e, $a_i, i = 0..m$, $b_j, j = 0..n$, 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, $R_k, k = 0..r$, 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):

$$a_1 : G_1 \rightarrow P_1, \ldots, a_m : G_m \rightarrow P_m$$

$$b_1 : H_1 \rightarrow Q_1, \ldots, b_n : H_n \rightarrow Q_n, n \geq 0$$

the following new transition is created:

$$e : G_1 \text{ and } \ldots \text{ and } G_m \text{ and } L_1 \text{ and } \ldots \text{ and } L_r \rightarrow$$

$$\{P_1; \ldots; P_m; \text{ if } H_1 \text{ then } Q_1; \ldots; \text{ if } H_n \text{ then } Q_n; R_1; \ldots; R_r\}$$

The modality ! forces the corresponding synchronized transition to be fireable.
Second case: $m = 0, r = 0$
a. $n > 1$
For each set of transitions (there may be several):

$$b_1 : H_1 \rightarrow Q_1, \ldots, b_n : H_n \rightarrow Q_n$$

the following new transition is created:

$$e : H_1 \text{ or } \ldots \text{ or } H_n \rightarrow \{ \text{ if } H_1 \text{ then } Q_1; \ldots; \text{ if } H_n \text{ then } Q_n\}$$

b. $n = 1$
For each set of transitions (there may be several):

$$b_1 : H_1 \rightarrow Q_1$$

the following new transition is created:

$$e : true \rightarrow \{ \text{ if } H_1 \text{ then } Q_1\}$$

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.

```
block FlattenedPrimaryPowerSupplySystem
  RepairableComponentState GR.s, Line1.TR1.s, Line2.TR2.s(init=WORKING);
  RepairableComponentState Line1.CBD1.s, Line1.CBU1.s(init=WORKING);
  RepairableComponentState Line2.CBD2.s, Line2.CBU2.s(init=WORKING);
  Boolean GR.inFlow(reset=true);
  Boolean outFlow, GR.outFlow, Line1.outFlow, Line2.outFlow(reset=false);
  Boolean Line1.TR1.inFlow, Line1.CBD1.inFlow, Line1.CBU1.inFlow(reset=false);
  Boolean Line2.TR2.inFlow, Line2.CBU2.inFlow, Line2.CBD2.inFlow(reset=false);
  Boolean Line1.TR1.outFlow, Line1.CBD1.outFlow(reset=false);
  Boolean Line2.TR2.outFlow, Line2.CBD2.outFlow(reset=false);
  Boolean Line1.CBU1.outFlow, Line2.CBU2.outFlow(reset=false);
  event GR.repair, GR.failure, Line2.CBD2.repair;
  event Line1.TR1.failure, Line1.TR1.repair, Line1.CBU1.failure;
  event Line1.CBD1.failure, Line1.CBD1.repair, Line2.TR2.failure;
  event Line2.CBU2.failure, Line2.CBU2.repair, Line2.CBD2.failure;
  event Line1.CBU1.repair, Line2.TR2.repair;
transition
  Line1.TR1.failure: Line1.TR1.s==WORKING -> Line1.TR1.s := FAILED ;
  Line1.TR1.repair: Line1.TR1.s==FAILED -> Line1.TR1.s := WORKING ;
  Line1.CBD1.failure: Line1.CBD1.s==WORKING -> Line1.CBD1.s := FAILED ;
  Line1.CBD1.repair: Line1.CBD1.s==FAILED -> Line1.CBD1.s := WORKING ;
  Line1.CBU1.failure: Line1.CBU1.s==WORKING -> Line1.CBU1.s := FAILED ;
  Line1.CBU1.repair: Line1.CBU1.s==FAILED -> Line1.CBU1.s := WORKING ;
  Line2.TR2.failure: Line2.TR2.s==WORKING -> Line2.TR2.s := FAILED ;
  Line2.TR2.repair: Line2.TR2.s==FAILED -> Line2.TR2.s := WORKING ;
  Line2.CBD2.failure: Line2.CBD2.s==WORKING -> Line2.CBD2.s := FAILED ;
  Line2.CBD2.repair: Line2.CBD2.s==FAILED -> Line2.CBD2.s := WORKING ;
  Line2.CBU2.failure: Line2.CBU2.s==WORKING -> Line2.CBU2.s := FAILED ;
  Line2.CBU2.repair: Line2.CBU2.s==FAILED -> Line2.CBU2.s := WORKING ;
  GR.failure: GR.s==WORKING -> GR.s := FAILED ;
  GR.repair: GR.s==FAILED -> GR.s := WORKING ;
assertion
  Line1.TR1.outFlow := (Line1.TR1.s==WORKING) and Line1.TR1.inFlow;
  Line1.CBD1.outFlow := (Line1.CBD1.s==WORKING) and Line1.CBD1.inFlow;
  Line1.CBU1.outFlow := (Line1.CBU1.s==WORKING) and Line1.CBU1.inFlow;
  Line1.CBU1.inFlow := GR.outFlow ;
  Line1.TR1.inFlow := Line1.CBU1.outFlow ;
  Line1.CBD1.inFlow := Line1.TR1.outFlow ;
  Line1.outFlow := Line1.CBD1.outFlow ;
  Line2.TR2.outFlow := (Line2.TR2.s==WORKING) and Line2.TR2.inFlow;
  Line2.CBD2.outFlow := (Line2.CBD2.s==WORKING) and Line2.CBD2.inFlow;
  Line2.CBU2.outFlow := (Line2.CBU2.s==WORKING) and Line2.CBU2.inFlow;
  Line2.CBU2.inFlow := GR.outFlow;
  Line2.TR2.inFlow := Line2.CBU2.outFlow;
  Line2.CBD2.inFlow := Line2.TR2.outFlow;
  Line2.outFlow := Line2.CBD2.outFlow;
  GR.outFlow := (GR.s== WORKING) and GR.inFlow;
  outFlow := Line1.outFlow or Line2.outFlow;
end
```

Figure 3.18: Flattened *Primary Power Supply* system

3.5.3 Hiding

Events involved in a synchronization continue to exist individually. However, if they must not occur individually, they can be hidden using the *hiding* mechanism: the keyword "hide" followed by the names of the events, separated by commas. Hidden events and the transitions they label are just removed from C_{flat} or from B_{flat}.

Example 3.3 (A Primary Power Supply system)**.** In the model given Figure 3.15 the synchronized transition "CCFTransformers" is added to the model depicted Figure 3.14. The resulting flattened model is the model presented Figure 3.18, in which the event "CCFTranformers" and the corresponding transition (see Figure 3.19) have been added. □

```
block PrimaryPowerSupplySystem
  ...
event CCFTransformers;
transition
  CCFTransformers : (Line1.TR1.s==WORKING) or (Line2.TR2.s==WORKING) ->
      { if Line1.TR1.s==WORKING then Line1.TR1.s = FAILED;
        if Line2.TR2.s==WORKING then Line2.TR2.s = FAILED;}
  ...
end
```

Figure 3.19: Flattened *Primary Power Supply* system: synchronizations

3.6 Discussion

3.6.1 About models reuse

The great advantage of high level modeling languages is the ability to reuse models of components or even models of systems, i.e. to capitalize knowledge. There are actually two quite distinct types of reuse:

1. The reuse of components, and
2. The reuse of modeling patterns.

They correspond to the respective strengths of object-oriented and prototype-oriented paradigms.

In the first case, libraries of reusable components (e.g. classes or functions) are defined. A reusable component is integrated as it is in the model. The notion of libraries of reusable components comes directly from programming languages, e.g. Qt library [18] or STL for C++. It made the success of modeling languages such as Matlab/Simulink and Modelica [81]: multiple libraries dedicated to a particular area are available. This type of models reuse corresponds to the object-oriented paradigm.

The reuse of modeling patterns follows another principle. The idea is to start from an existing code, to duplicate it and to adapt it to particular needs. The famous "design patterns" [40] are the epitome of this approach. It naturally corresponds to the prototype-oriented paradigm.

Experience shows that for AltaRica the reuse of modeling patterns is really advantageous (see e.g. [57]). Although the definition of libraries of reusable components can be also beneficial.

3.6.2 About parametric models and scripts

One often has to adapt a generic component to a particular need. For example, in the model of the repairable component defined in Figure 3.7, one may want to define a failure rate and a repair rate

of the component (they will be named lambda and mu). These rates will be given a value at the instantiation of the component.

AltaRica 3.0 introduces the notion of parameter (directly inspired from Modelica). A parameter has a default value that can be changed when the class is instantiated. The use of parameters is illustrated Figure 3.20. In this example we define a parameter "lambda" inside the class "RepairableComponent". This parameter has a default value, which is equal to $1.0e-3$. When the class "Transformer" which extends the class "RepairableComponent" is instantiated inside the block "Line1", the value of the parameter "lambda" is set to $2.34e-5$.

```
class RepairableComponent
  ...
  event failure (delay = exponential(lambda));
  parameter Real lambda = 1.0e-3;
  ...
end
block PrimaryPowerSupplySystem
  ...
  block Line1
    Transformer TR1(lambda = 2.34e-5);
    ...
  end
  ...
end
```

Figure 3.20: Illustration of *parameters* usage

Another need of generic components is the ability to define components with a variable number of inputs (or outputs). For example, imagine that one needs to define a repairable component with a variable number of Boolean inputs and one Boolean output which is equal to the disjunction of inputs if the component is operational and false if it is failed. It is obviously preferable to avoid defining such a component with two inputs, another one with three inputs and so on. Ideally, we would like to define a generic component with a variable number of inputs and set this number at the instantiation.

In Figaro modeling language (see e.g. [17]) the design of such generic components was implemented by means of quantifiers. In Modelica (see e.g. [37]) it can be done using arrays.

A general solution is to use a scripting language, integrated in the modeling language, to automatically generate models. This is the conclusion of our work on comparison between AltaRica and PEPA nets for modeling systems with mobile components ([59]). No decision has been taken yet on this subject for AltaRica 3.0.

3.6.3 About graphical representation of models

AltaRica 3.0 is a textual language. However graphical representations can be associated with textual models. They are of great interest for several reasons:

- Graphical representations offer a convenient way to create and edit models.
- They help to better understand models and to communicate.
- They can be used to animate models, i.e. to graphically visualize the simulation of AltaRica 3.0 models.
- They allow navigation in the model, etc.

From our point of view,

1. First, it is necessary to distinguish between graphical editing and graphical simulation of models. Different graphical representations, technologies and tools can be used in each of these cases.

2. Second, it is necessary to distinguish between the model and its graphical representation. Graphical representations are only partial views of the model.

We have recently shown in [83] that graphical simulation of models can be implemented using a graphical modeling language and the corresponding simulator. The idea is that the graphical simulator and the AltaRica 3.0 simulator communicate via the exchange of variables value. Graphical animations are totally independent from AltaRica 3.0 models. Also the language for graphical representation and animation of models can be coupled with other tools (not necessarily dedicated to AltaRica). The immediate benefit is that the behavioral model can be designed independently from its graphical representation for animation, which can be created later.

The desire to have a complete correspondence between the model and its graphical representation results either in the significant reduction of the expressive power of the modeling language or in the considerably more complex graphical representations to make them lose their interest.

Therefore, AltaRica 3.0 model cannot have a unique graphical representation. But several different graphical representation can be associated with textual models. Each representation gives a partial view of the model. The idea is that these views can be generated automatically. Graphical representations of AltaRica models may be of different nature:

- Structural diagrams, representing only the structural part of the model. This type of diagrams is used in all AltaRica workshops.

- Automaton diagrams, representing the behavioral part of the model, as the one depicted Figure 3.7. The corresponding AltaRica code is given Figure 3.8.

- Sequence diagrams to represent synchronizations.

Structural diagrams can be of different form. At least, we can cite four graphical representations for hierarchical models:

Planar representation: "Process & Instrumentation Diagram" scheme, represents the structure as nested boxes connected together by wires. See for example Figure 3.1 as an illustration. It represents not only the hierarchy, but also the "horizontal" links between components. This is the most common type of representation. It is implemented in all AltaRica workshops and also in Modelica or Matlab Simulink workshops. SysML Internal Block Diagram is based on this type of representation. However, it seems difficult to automatically generate planar representation as pointed out by Fuhrmann [39].

Tree representation: Makes clear the hierarchical structure of the model. The tree representation of the *Power Supply System* is given Figure 3.21. Blocks declaration can be folded/unfolded just as in planar representation. Conversely to the planar representation, only "vertical" relations between components can be represented using this type of diagram. Fault Trees and SysML Block Definition diagrams are based on this type of graphical representation. It is possible to automatically generate this representation from textual models, like it is done for example for Fault Trees.

1D representation: Explorer like representation. It gives a hierarchical view of the model structure. The 1D representation of the *Power Supply System* is depicted Figure 3.22. Each level can folded/unfolded like in navigation panel of Windows explorer. Like *Tree representation*, it cannot be used to represent links between components. It is possible to automatically generate this representation from textual models.

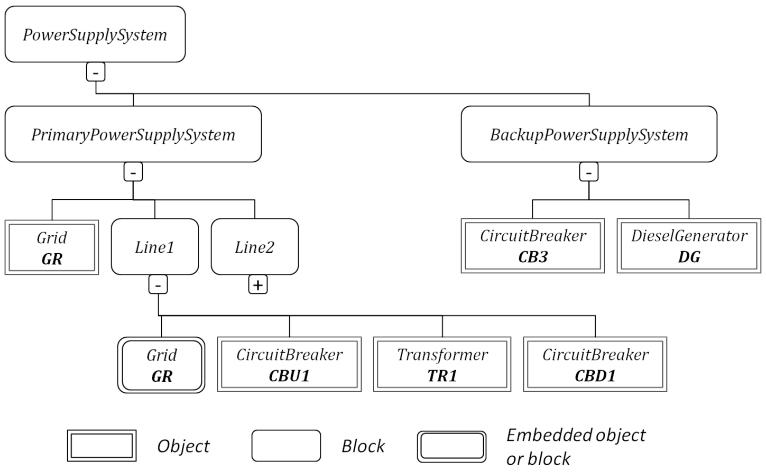


Figure 3.21: Tree representation of the *Power Supply System*

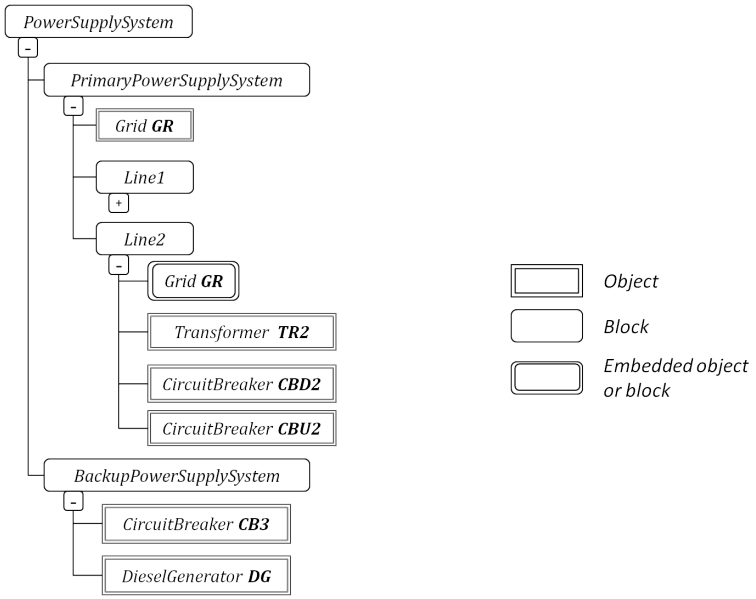


Figure 3.22: 1D representation of the *Power Supply System*

Tabular representation: Is a simplified textual representation that makes it possible to edit the model via a spreadsheet. The tabular representation of the *Power Supply System* is given Figure 3.23.

name	type	child 1	child 2	child3
Power Supply System	block	Primary Power Supply System	Backup Power Supply System	
Primary Power Supply System	block	GR	Line1	Line2
Backup Power Supply System	block	DG	CB3	
GR	instance	GRID		
DG	instance	DieselGenerator		
..	...	..	..	...

Figure 3.23: Tabular representation of the *Power Supply System*

These graphical views of model structure can be used to visualize and also to edit the model. Anyway, textual models should be independent from graphical representations, like CSS (Cascading Style Sheets) are independent from HTML.

Summary

In this chapter we presented structural constructs of AltaRica 3.0. AltaRica 3.0 introduces two concepts to build hierarchical models: *classes* and *blocks*. *Blocks* represent components having a unique occurrence in the model. The concept of *block* comes from prototype-oriented programming languages. *Classes* define generic components. They are used in the model via instantiation. The concept of *class* comes from object-oriented programming languages. Models can be organized in hierarchies of components by means of three operations: composition, inheritance and aggregation.

AltaRica 3.0 makes a clear distinction between:

- the stabilized knowledge, which is incorporated into libraries of "on-the-shelf" components, represented by *classes*, and
- the "sandbox" in which the analyst is designing his model of the system under study. In the sandbox some components are unique; some others are instances of reusable components.

AltaRica 3.0 modeling language is, in fact, the combination of its underlying mathematical formalism, Guarded Transition Systems (GTS), presented in chapter 2, and the paradigm to structure models, System Structure Modeling Language (S2ML), introduced in this chapter:

$$\text{AltaRica 3.0} = \text{S2ML} + \text{GTS}$$

Each hierarchical AltaRica 3.0 model can be flattened into a unique GTS according to the flattening rules defined in this chapter.

In the next chapter we will see how to compile GTS into Fault Trees and critical sequences of events.

Chapter 4

Compilation into Fault Trees or critical sequences of events

Guarded Transition Systems (or AltaRica 3.0 models) can be compiled into Fault Trees or critical sequences of events. The compilation into Fault Trees is of interest for several reasons: first, automatically generating Fault Trees from high-level models is easier and less time consuming rather than creating them from scratch; second, high level models greatly improve the design, the sharing and the maintenance of models; finally, assessment tools for Boolean models are much more efficient than those for states/transitions models. In general, the price to pay is the loss of sequencing among events: sequences of events are compiled into conjuncts of events. However, if the GTS is combinatorial, its compilation to Fault Trees is efficient and does not loose information. Many real-life models are relatively simple extensions of Reliability Block Diagrams and, thus, can be compiled efficiently into Fault Trees. When the events of the model are not statistically independent, it is more appropriate to generate critical sequences of event rather than Fault Trees.

The goal of this chapter is to present the principle of the algorithm to compile Guarded Transition Systems into Fault Trees and critical sequences of events. This algorithm extends the algorithm, proposed in [88] for Mode Automata (or AltaRica Data-Flow) to GTS. It is based on the advanced partitioning techniques that ensure its efficiency for some categories of models.

This chapter is organized as follows. Section 4.1 presents a motivating example that is used as a red wire in this chapter. Section 4.2 is dedicated to related works. Section 4.3 describes the algorithm of compilation of Guarded Transition Systems into Fault Trees and critical sequences of events. In Section 4.4, we extend the algorithm presented in the previous section to Timed/Stochastic Guarded Transition Systems. Finally, Section 4.5 discusses the complexity of the presented algorithm and its correctness.

4.1 Motivations

Fault Trees are probably the most popular formalism to support Probabilistic Risk and Safety Analyses. Efficient algorithms have been designed to assess these models (see e.g. [93, 91]) and mature commercial tools are now available. Despite their interest, Fault Trees suffer from the severe drawback to be very far from the specifications of the system under study. They rely on a great deal of implicit knowledge and expertise of the analyst. As a consequence, they are hard to share amongst the stakeholders and to maintain throughout the life cycle of systems. A small change in the specifications of a system may require revisiting the Fault Trees designed for that system, which is both resource consuming and error prone.

AltaRica modeling language has been created to tackle this problem. An algorithm to compile AltaRica Data-Flow (or Mode Automata) to Fault Trees has been proposed in [88]. The compilation

to Fault Trees and critical sequences of events of the first version of AltaRica, based on constraint automata [80, 7], also called here AltaRica LaBRI, has been reported in [43].

AltaRica 3.0 is a new version of the language [82]. Its new underlying mathematical model, Guarded Transition Systems [90, 84], improves the expressive power of the previous versions by introducing a fixpoint mechanism to stabilize the values of the flow variables after each transition firing. This mechanism allows the design of acausal components and the treatment of systems with instant loops. It makes the generation of Fault Trees more complex. Nevertheless, we show here that this generation can remain efficient thanks to advanced partitioning techniques.

In order to illustrate the different steps of the algorithm, we propose to study a Data Gathering and Processing network. This system is interesting because it contains acausal components and instant loops.

Example 4.1 (A Data Gathering and Processing Network)**.** Consider the network depicted Figure 4.1, which is inspired from [94]. This network is made of:

- Three workstations W1, W2, and W3 producing data;
- Two processing units P1 and P2 in charge of processing data;
- Six switches SW1,..., SW6 receiving data from workstations and/or other switches and transmitting them to processing units or other switches.

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

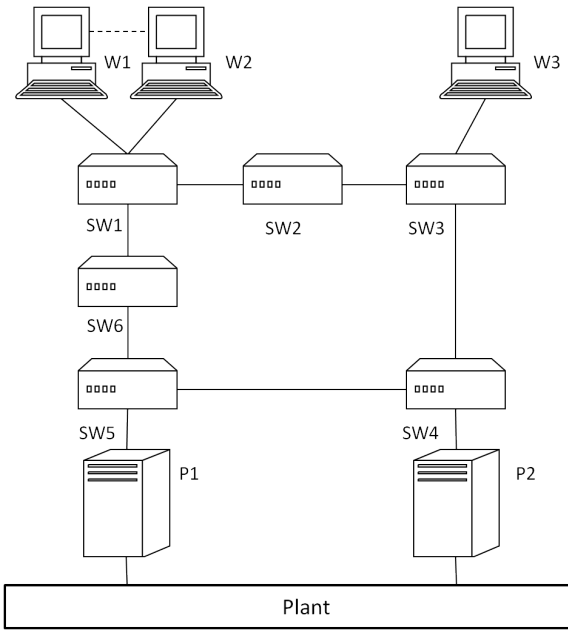


Figure 4.1: A Data Gathering and Processing Network

All components may fail in operation. Moreover, workstations may have a common cause failure. Failure rates are given in Table 4.1.

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

Table 4.1: Failure rates of components of the network

Component	Failure rate
Switch	$10^{-4}\ 1/h$
Workstation	$10^{-5}\ 1/h$
Processing Unit	$10^{-4}\ 1/h$

- Processing unit P1 cannot send data to the plant.
- Processing unit P2 cannot send data to the plant.
- Neither P1 nor P2 can send data to the plant.

We shall use this network as a red wire example to describe the different steps of the compilation of Guarded Transition Systems into Fault Trees.

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

AltaRica 3.0 model of the Data Gathering and Processing network

Modeling non-repairable components Since all the components of the described system are non-repairable we define a class representing a non-repairable component:

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

Models of other components will extend this class.

Modeling processing units Processing units are represented by the following class:

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

This class extends the class `NonRepairableComponent`. Its assertion states that if the processing unit is working, it receives data in input, process them, and emits data as output. If the processing unit is failed, it does not process, nor output any data.

Modeling workstations The model of the workstations is different from the model of the non-repairable component, because we should take into account the fact that the workstation W2 is in standby mode. The class representing workstations is as follows:

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

In the initial state, the workstation is in standby mode. The event `start` makes it change its state to working. When the workstation is operational, it produces data (its flow variable `outFlow` is true); otherwise, it does not output any data (`outFlow` is equal to false).

Modeling switches Models of processing units and workstations are directional (Data-Flow): their inputs and outputs and the direction of the propagation of flows is clearly identified. The model of the switch is more interesting as it is acausal.

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

In the above model, when the switch is working it may receive data from workstations via the flow variable `inFlow`, but also from its neighbors (the other switches) via the flow variables `leftFlow` and `rightFlow`. If the switch is working, the flows linking it to its neighbors are all true as soon as one of them is true. If the switch is failed, then there is no relation amongst them.

Modeling the whole system The model of the whole system is given Figure 4.2. In this model the operator :=: is used to represent bidirectional connections (see definition in Section 2.4) between switches: data can circulate both ways between them. The direction of the flow is determined at run time for it depends on the global state of the system.

Observers: Observers are like flow variables, except they cannot be used in transitions and assertions. They are quantities to be observed by the assessment tools. In the model, we define three observers: `P1failed`, `P2failed` and `P1P2failed` corresponding to the three events that we want to observe for the network.

Synchronizations: Synchronizations are used to compel several events to occur at the same time

(see section 2.4 for more details about synchronizations). Here synchronizations are used to represent the fact that the workstation W2 starts when the workstation W1 is failed (transition `W1_failure`) and also to model the common cause failure of the workstations (transition `ccf`). Events involved in a synchronization continue to exist individually. However, it is possible to hide the events that should not occur independently of a synchronization: this is precisely the case of the events `W2.start` and `W1.failure`.

```
block Network
  Workstation W1, W3;
  Workstation W2(s.init = STANDBY);
  Switch SW1, SW2, SW3, SW4, SW5, SW6;
  ProcessingUnit P1, P2;
  parameter Real lambda = 5e-6;
  event ccf(delay = exponential(lambda));
  event W1_failure(delay = exponential(W1.lambda));

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

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

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

Figure 4.2: AltaRica 3.0 model of the Data Gathering and Processing Network: main block

4.2 Related Works

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

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

4.2.1 Algorithms based on backward analysis

In this category, most of the proposed approaches rely on various extensions of Reliability Block Diagrams (see e.g. [62]). Basic blocks of the diagram carry out both failures and inputs/outputs relations. The flow circulating in the diagram is analyzed backward to generate the Fault Tree. This idea stems from Artificial Intelligence tools such as Assumption-based Truth Maintenance Systems [28].

A similar idea is used in the HiP-HOPS workbench [78, 75]. This workbench enables to add reliability data to models imported from different modeling tools such as Matlab/SIMULINK, Eclipse-based UML tools, etc. It then enables to automatically generate Fault Trees and FMEA tables. The underlying formalism of Hip-HOPS is also an extension of Reliability Block Diagrams, in which the system is described by hierarchies of blocks and the outputs of the blocks are written as a discrete function of internal failures and inputs. AltaRica 3.0 generalizes this kind of models and the algorithm described in this chapter is very efficient on them.

The same principle is used in [54], where Fault Trees are automatically generated from AADL models.

4.2.2 Algorithms based on fault injection

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

In this category, we can find the algorithm used in KB3 workbench [15], developed by EDF R&D, where Fault Trees are automatically generated from Figaro models [17]. The proposed algorithm is based on the exploration of all the possible combinations of failure events in order to determine if they lead to the system failure. Different truncation criteria are applied, such as the maximum number of events in a sequence, the probability of the sequence, etc.

The same principle is used in [20], where M. Bozzano et al. use NuSMV for Fault Tree analysis and apply different symbolic model-checking techniques in order to improve the efficiency of the algorithm.

In [19], M. Bozzano et al. translate AltaRica Data-Flow models, created within Cecilia OCAS workbench, into NuSMV input format and apply algorithms from [20] to automatically generate Fault Trees. They also compare the efficiency of NuSMV with the sequence generator of AltaRica Data-Flow used in Cecilia OCAS workbench. The sequence generator of AltaRica Data-Flow basically explores all the possible combinations of events that lead the system from its nominal state to failure state. The truncation criterion is the number of events involved in a sequence.

In [43], A. Griffault et al. describe algorithms for the automatic generation of Fault Trees and critical sequences of events from AltaRica LaBRI [80, 7] models using symbolic model-checking techniques.

As seen in chapter 2, GTS generalize Reliability Block Diagrams. However, it is essentially a formalism to describe (finite) state machines and to compose them. It is therefore more expressive than all extensions of Reliability Block Diagrams proposed so far. For instance, it is possible to model Common Cause Failures or shared resources amongst different blocks by synchronizing events. Because of its expressiveness, GTS cannot be compiled into Fault Trees just by backward induction on the values of the flow variables. More elaborated compilation schemes must be applied. Fortunately, thanks to advanced partitioning techniques, we do not need to explore the whole system model in order to discover sequences of events leading the system from its nominal state to its failure state.

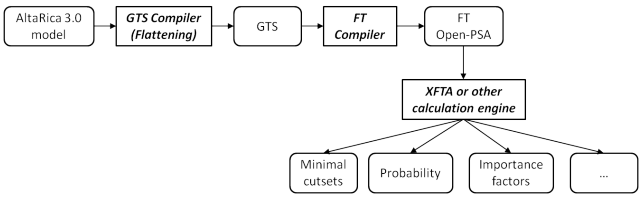


Figure 4.3: Fault Tree Analysis with AltaRica 3.0 models

4.3 Compilation algorithm

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

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

The whole assessment process is illustrated in Figure 4.3:

- First, AltaRica 3.0 model is flattened into a Guarded Transition System (GTS) according to the rules defined in Section 3.5.
- Then, the GTS generated by the previous step is compiled into a Fault Tree which is exported in Open-PSA format [51].
- Finally, the generated Fault Tree is assessed with any calculation engine supporting this format, in order to calculate minimal cutsets, probabilities of the top events, importance factors, etc. For instance, the calculation engine XFTA [91] can be used for this purpose.

From now, assume that AltaRica 3.0 model has been flattened into a GTS $G = \langle V = S \uplus F, E, T, \iota, A \rangle$ according to the rules defined in Section 3.5. The algorithm of compilation of G into Fault Trees or sequences of events works in 4 steps (see Figure 4.4):

Step 1: First, a given GTS G is partitioned into one or more independent GTS (see definition 2.5) and an independent assertion.

Step 2: Second, the reachability graphs of each independent GTS are calculated.

Step 3: Then, each reachability graph is separately compiled into Boolean equations or sequences of events.

Step 4: Finally, the independent assertion is compiled into Boolean equations.

Each of these steps is described in details in the following.

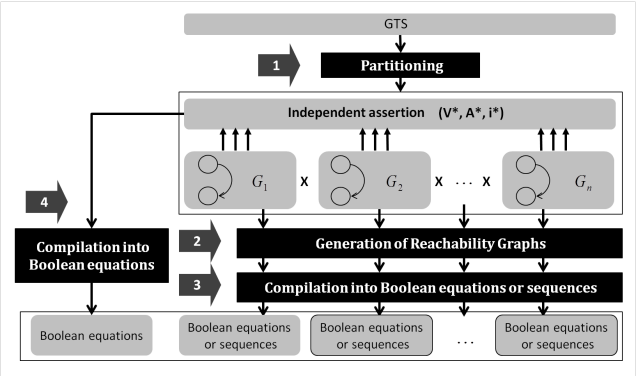


Figure 4.4: Compilation of GTS into Fault Trees or event sequences

4.3.1 Compilation of labeled Kripke Structures into Boolean formulae

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

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

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

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

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

In case of the generation of critical sequences of events the operator $\otimes$ is interpreted as a concatenation of sequences and the operator $\sum$ denotes sets of sequences.

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

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

4.3.2 Partitioning

In general, components of a system fail in a relatively independent way. To ensure the efficiency of the compilation algorithm, we should take advantage of this independence.

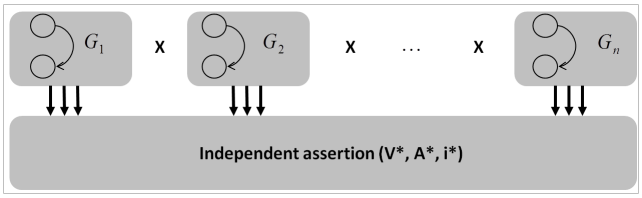


Figure 4.5: Partitioning of GTS

Definition 4.1 (Partition of a GTS)**.** Partitioning of a GTS $G = \langle V, E, T, A, \iota \rangle$ consists in representing G in the following way (see Figure 4.5 as an illustration):

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

where

- $G_i = \langle V_i, E_i, T_i, A_i, \iota_i \rangle$, $i = 1..n$ are independent Guarded Transition Systems in the sense of the definition 2.5,
- $\times$ denotes the free product of GTS (see definition 2.6) and
- $\langle V^*, A^*, \iota^* \rangle$ is an independent assertion, also called a glue,

such that

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

The last part (the glue) does not contain any behavior. Variables in V^* are only flow variables, they depend on the state variables and the flow variables of independent GTS G_i.

Partitioning is the key point of the algorithm that ensures its efficiency. A similar idea can be found in [42].

To partition a Guarded Transition System, one needs to analyze dependencies between its transitions in order to divide them into independent groups. That is why, variables involved in each transition, i.e. variables used in the guard and in the action of the transition, should be considered.

Definition 4.2 (Variables of a transition)**.** Let $G = \langle V, E, T, A, \iota \rangle$ be a GTS and let $t = \langle e, G, P \rangle$ be one of its transitions. We denote by $var(t)$ variables involved in the transition t:

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

where V' are variables, such that variables from $var(G) \cup var(P)$ depend on them via the assertion A (see Definition 2.14):

$$V' = \{v \in V : \exists u \in var(G) \cup var(P) \mid u \text{ depends on } v \text{ in } A\}$$

Definition 4.3 (Independence of transitions)**.** Let $G = \langle V, E, T, A, \iota \rangle$ be a GTS and let t_1 and t_2 be two transitions of G. We say that two transitions t_1 and t_2 are independent if they do not share any variables: $var(t_1) \cap var(t_2) = \emptyset$.

The last relation enables to divide transitions and, as a consequence, events and variables into independent sets T_i, E_i, V_i.

In practice, we consider an undirected graph G_T with nodes labeled by the transitions of G. There is an edge between two nodes of the graph G_T if the transitions labeling the nodes are dependent, i.e. $var(t_i) \cap var(t_j) \neq \emptyset$, $t_i, t_j \in T$. The connected components of G_T give us a partition of the transitions and, therefore, of the variables and the events:

$$T = T_1 \uplus T_2 \uplus \ldots \uplus T_n, T_i \cap T_j = \emptyset \ \forall \ i \neq j$$

$$E = E_1 \uplus E_2 \uplus \ldots \uplus E_n, E_i \cap E_j = \emptyset \ \forall \ i \neq j$$

$$V = V_1 \uplus V_2 \uplus \ldots \uplus V_n \uplus V^*, V_i \cap V_j = \emptyset \ \forall \ i \neq j$$

From the previous step, we have $G_i = \langle V_i, E_i, T_i, \iota_i, A_i \rangle$, $i = 1..n$, with $A_i = skip \ \forall i = 1..n$, and $\langle V^*, A, \iota^* \rangle$. We say that a flow variable $v \in V^*$ belongs to V_i iff the following holds:

$$\exists \ w \in V_i, \text{ such that } v \text{ depends on } w \text{ in } A \text{ and } \nexists \ u \in \bigcup_{j \neq i} V_j, \text{ such that } v \text{ depends on } u \text{ in } A.$$

In other words, a flow variable belongs to a partition G_i if it depends on variables from this partition and it does not depend on variables from other partitions.

In practice, the dependency graph $G_D[A]$ of the assertion A (see definition 2.15) enables to detect the flow variables, belonging to each independent GTS and to partition the assertion A into independent parts A_i according to the criterion given above. The remaining flow variables and instructions from A constitute the independent assertion $< V^*, A^*, \iota^* >$. Note that the instruction A^* is built over variables from V.

Example 4.2 (A Data Gathering and Processing Network)**.** In the model of the Network System, the transition `ccf` involves the variables `W1.s`, `W2.s` and `W3.s`. Since the transitions `ccf`, `W1_failure`, `W2.failure`, `W3.failure` share variables, used in their guards and actions, they belong to the same partition. Since the flow variables `Wi.outFlow`, $i = 1..3$, and `SWi.inFlow`, $i = 1, 3$ depend only on the variables from this partition, they also belong to it. Other transitions `SW1.failure`, ..., `SW6.failure`, `P1.failure`, `P2.failure` are independent from each other and belong to different partitions. So the partitioned Network System contains 9 independent GTS and an independent assertion. They are represented by blocks in Figure 4.6.

□

Remark 4.1. *Partitioning is a purely syntactic operation. Note that generally the partitioned GTS does not correspond to the structure of the initial AltaRica 3.0 model. It is particularly the case of the example given above. Only in the case of completely independent components, the partition corresponds to the structure of the AltaRica 3.0 model.*

4.3.3 Reachability Graph generation

For each independent Guarded Transition System $G = \langle V, E, T, \iota, A \rangle$ its reachability graph $\Gamma = (\Sigma, \Theta)$ is constructed according to the definition given in Section 2.5. Starting from the initial state $\sigma_0 = Propagate(A, \iota, \iota)$, calculated using the initial variable assignment ι and the assertion A, other states are discovered by firing step by step all transitions, fireable in the current state: if $\sigma \in \Sigma$ and there is a transition $t = \langle e, G, P \rangle$ which is fireable in σ, then $\tau = Fire(t, A, \iota, \sigma) \in \Sigma$ and $(\sigma, e, \tau) \in \Theta$.

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

Figure 4.6: Partitioned GTS representing the Data Gathering and Processing Network

Example 4.3 (A Data Gathering and Processing Network)**.** The reachability graph of the block `Part1` is depicted Figure 4.7. It contains 6 nodes (or states) S0, ..., S5, labeled by variable assignments. Its transitions are labeled by the events of the block `Part1`: `ccf`, `W1_failure`, `W2.failure` and `W3.failure`. We shall use this graph in order to illustrate how a reachability graph is compiled into Boolean formulae.

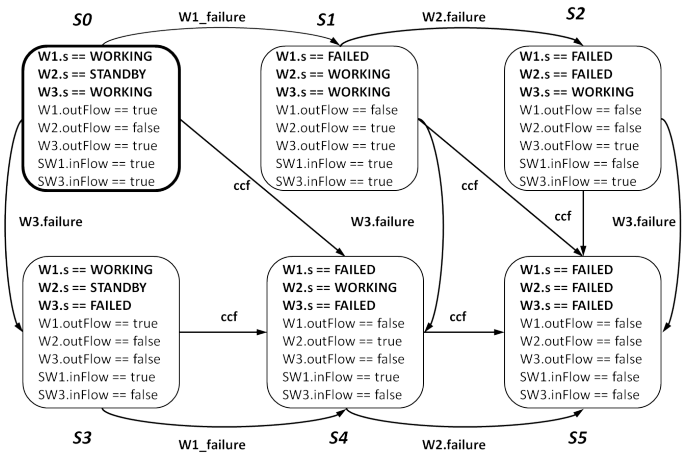


Figure 4.7: Reachability graph of workstations

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

4.3.4 Compilation of Reachability Graphs

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

The principle of compilation of Γ is given in Section 4.3.1. Here we explain in details the algorithms of:

- compilation of Γ into Fault Trees.
- generation of sequences of events from Γ.

We illustrate them on the red wire example of this chapter.

Compilation into Fault Trees

The algorithm captures failure scenarios into a set of Boolean equations. It produces a Boolean formula $\phi_{(v,c)}$ for each pair (v, c), where v is a variable from V and c is its value, $c \in dom(v)$, such that the variables of $\phi_{(v,c)}$ are events from E.

Note that if $\{e_1, \ldots, e_k\}$, $e_j \in E$ $\forall j = 1..k$, is a minimal cutset of $\phi_{v,c}$, then there is a path in the reachability graph Γ, such that $(\sigma_0, e_1, \sigma_1) \in \Theta, (\sigma_1, e_2, \sigma_2) \in \Theta, \ldots, (\sigma_{k-1}, e_k, \sigma_k) \in \Theta$, and $\sigma_k(v) = c$.

In order to avoid conflicts raised by the composition of components (for more details, see [88]), we need to consider not only events occurring along the path π but also those that do not. Finally, the algorithm works as follows:

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

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

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

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

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

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

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

In practice, the reachability graph is created and compiled into Boolean formulae at the same time according to the algorithm given Figure 4.8. The algorithm comes from [88] and is adapted to Guarded Transition Systems.

```
C ← {⟨σ0, 0⃗⟩}, D ← ∅
while C ≠ ∅ do
  Let ⟨σ, P⃗⟩ ∈ C
  C ← C \ {⟨σ, P⃗⟩}, D ← D ∪ {⟨σ, P⃗⟩}
  forall t = ⟨e, G, Q⟩ ∈ T, σ(G) = true do
    τ = Fire(Q, A, ι, σ)
    C ← C ∪ {⟨τ, P⃗[e] ← 1⟩}
  done
done
```

Figure 4.8: The algorithm to compile a GTS into Boolean expressions

Example 4.4 (A Data Gathering and Processing Network)**.** In order to illustrate this algorithm, consider the reachability graph given Figure 4.7. Boolean equations associated with each node (or state) of the this graph (step 3 of the algorithm) are the following:

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

- $\phi_{S5} = ccf \bigwedge \overline{W1\_failure} \bigwedge \overline{W2.failure} \bigwedge \overline{W3.failure} \bigvee$
  $ccf \bigwedge \overline{W1\_failure} \bigwedge W2.failure \bigwedge \overline{W3.failure} \bigvee$
  $ccf \bigwedge \overline{W1\_failure} \bigwedge \overline{W2.failure} \bigwedge W3.failure \bigvee$
  $ccf \bigwedge \overline{W1\_failure} \bigwedge W2.failure \bigwedge W3.failure \bigvee$
  $ccf \bigwedge W1\_failure \bigwedge \overline{W2.failure} \bigwedge \overline{W3.failure} \bigvee$
  $ccf \bigwedge W1\_failure \bigwedge W2.failure \bigwedge \overline{W3.failure} \bigvee$
  $ccf \bigwedge W1\_failure \bigwedge \overline{W2.failure} \bigwedge W3.failure \bigvee$
  $\overline{ccf} \bigwedge W1\_failure \bigwedge W2.failure \bigwedge W3.failure$

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

$\phi_{(W1.s,WORKING)} = \phi_{S0} \bigvee \phi_{S3}$	$\phi_{(W1.s,FAILED)} = \phi_{S1} \bigvee \phi_{S2} \bigvee \phi_{S4} \bigvee \phi_{S5}$
$\phi_{(W2.s,STANDBY)} = \phi_{S0} \bigvee \phi_{S3}$	$\phi_{(W2.s,WORKING)} = \phi_{S1} \bigvee \phi_{S4}$
$\phi_{(W2.s,FAILED)} = \phi_{S2} \bigvee \phi_{S5}$	
$\phi_{(W3.s,WORKING)} = \phi_{S0} \bigvee \phi_{S1} \bigvee \phi_{S2}$	$\phi_{(W3.s,FAILED)} = \phi_{S3} \bigvee \phi_{S4} \bigvee \phi_{S5}$
$\phi_{(W1.outFlow,true)} = \phi_{S0} \bigvee \phi_{S2}$	$\phi_{(W1.outFlow,false)} = \phi_{S1} \bigvee \phi_{S3} \bigvee \phi_{S4} \bigvee \phi_{S5}$
$\phi_{(W2.outFlow,true)} = \phi_{S1} \bigvee \phi_{S3}$	$\phi_{(W2.outFlow,false)} = \phi_{S0} \bigvee \phi_{S2} \bigvee \phi_{S4} \bigvee \phi_{S5}$
$\phi_{(W3.outFlow,true)} = \phi_{S0} \bigvee \phi_{S1} \bigvee \phi_{S4}$	$\phi_{(W3.outFlow,false)} = \phi_{S2} \bigvee \phi_{S3} \bigvee \phi_{S5}$
$\phi_{(SW1.inFlow,false)} = \phi_{S4} \bigvee \phi_{S5}$	$\phi_{(SW1.inFlow,true)} = \phi_{S0} \bigvee \phi_{S1} \bigvee \phi_{S2} \bigvee \phi_{S3}$
$\phi_{(SW3.inFlow,true)} = \phi_{S0} \bigvee \phi_{S1} \bigvee \phi_{S4}$	$\phi_{(SW3.inFlow,false)} = \phi_{S2} \bigvee \phi_{S3} \bigvee \phi_{S5}$

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

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

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

□

Generation of sequences of events

In the case of generation of critical sequences of events the operator $\otimes$ is interpreted as a concatenation of sequences and the operator $\sum$ denotes sets of sequences. The operator of concatenation will be further noted as $\cdot$. We search for all the paths from the state σ_0 to other states of the graph. Events occurring along a path π are transformed into a concatenation of events. First, we associate with each state of the graph a disjunction of sequences obtained by the compilation of paths. Then, we associate with each couple (variable, value) a disjunction of sequences associated with each state, where this variable takes this value.

Example 4.5 (A Data Gathering and Processing network)**.** In order to illustrate this algorithm, consider the reachability graph given Figure 4.7. Sequence generated for each state of this graph are the following:

- $\phi_{S0} = \epsilon$
- $\phi_{S1} = W1\_failure$
- $\phi_{S2} = W1\_failure \cdot W2.failure$
- $\phi_{S3} = W2.failure$

- $\phi_{S4} = ccf + W3.failure \cdot ccf + W3.failure \cdot W1\_failure + W1\_failure \cdot W3.failure$
- $\phi_{S5} = ccf \cdot ccf + W1\_failure \cdot ccf + ccf \cdot W2.failure +$
  $+W3.failure \cdot W1\_failure \cdot W2.failure + W1\_failure \cdot W3.failure \cdot W2.failure +$
  $+W3.failure \cdot W1\_failure \cdot ccf + W1\_failure \cdot W3.failure \cdot ccf +$
  $+W1\_failure \cdot W2.failure \cdot W3.failure + W3.failure \cdot ccf \cdot W2.failure +$
  $+W3.failure \cdot ccf \cdot ccf + W1\_failure \cdot W2.failure \cdot ccf$

Sequences associated with each pair (variable, value) are as follows:

$\phi_{(W1.s,WORKING)} = \phi_{S0} + \phi_{S3}$	$\phi_{(W1.s,FAILED)} = \phi_{S1} + \phi_{S2} + \phi_{S4} + \phi_{S5}$
$\phi_{(W2.s,STANDBY)} = \phi_{S0} + \phi_{S3}$	$\phi_{(W2.s,WORKING)} = \phi_{S1} + \phi_{S4}$
$\phi_{(W2.s,FAILED)} = \phi_{S2} + \phi_{S5}$	
$\phi_{(W3.s,WORKING)} = \phi_{S0} + \phi_{S1} + \phi_{S2}$	$\phi_{(W3.s,FAILED)} = \phi_{S3} + \phi_{S4} + \phi_{S5}$
$\phi_{(W1.outFlow,true)} = \phi_{S0} + \phi_{S2}$	$\phi_{(W1.outFlow,false)} = \phi_{S1} + \phi_{S3} + \phi_{S4} + \phi_{S5}$
$\phi_{(W2.outFlow,true)} = \phi_{S1} + \phi_{S3}$	$\phi_{(W2.outFlow,false)} = \phi_{S0} + \phi_{S2} + \phi_{S4} + \phi_{S5}$
$\phi_{(W3.outFlow,true)} = \phi_{S0} + \phi_{S1} + \phi_{S4}$	$\phi_{(W3.outFlow,false)} = \phi_{S2} + \phi_{S3} + \phi_{S5}$
$\phi_{(SW1.inFlow,false)} = \phi_{S4} + \phi_{S5}$	$\phi_{(SW1.inFlow,true)} = \phi_{S0} + \phi_{S1} + \phi_{S2} + \phi_{S3}$
$\phi_{(SW3.inFlow,true)} = \phi_{S0} + \phi_{S1} + \phi_{S4}$	$\phi_{(SW3.inFlow,false)} = \phi_{S2} + \phi_{S3} + \phi_{S5}$

Sequences generated for other reachability graphs are quite similar. For example,

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

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

4.3.5 Compilation of the independent assertion

Let us denote by U the set of variables from independent GTS:

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

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

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

In order to compile the assertion into Boolean formulae efficiently, one need to separate it into independent parts. Here we use the same technique as for the optimization of the assertion A, presented in section 2.6. Consider $G_D[A^*]$ - the dependency graph of the assertion A^* (see Definition 2.15). This graph contains cycles. The strongly connected components of $G_D[A^*]$ divide variables of A^* into sets $W_i, i = 1..m$ and enable to decompose the assertion A^* into blocks of instructions $A^*_i, i = 1..m$, where m is the number of strongly connected components of $G_D[A^*]$:

$$A^* = A^*_1; A^*_2; \ldots; A^*_m$$

The algorithm calculates $\Phi = \{\phi_{(v,c)}, v \in V, c \in dom(v)\}$, the set of Boolean equations, and $D = \{\langle v, dom(v) \rangle, v \in V\}$, the set of pairs that associated to each variable v its domain $dom(v)$. It works as follows:

```
Φ = ∅, Φ ← Φ ∪ {φ_(u,c), u ∈ U, c ∈ dom(u)}
D = ∅, D ← D ∪ {⟨u, dom(u)⟩, u ∈ U}
CompileAssertion2BooleanExpressions(V*, A*, ι*, Φ, D)
  BuildDependencyGraph(A*, G_D);
  BuildStronglyConnectedComponents(G_D, G_D^SCC);
  C = ∅, C ← C ∪ V*
  while C ≠ ∅ do
    Let f ∈ C
    ⟨f, dom(f)⟩ ← ComputeVariableDomain(f, G_D^SCC, C, Φ, D);
  done
```

The function `ComputeVariableDomain(`f`,` G_D^{SCC}`,` C`,` Φ`,` D`)` calculates recursively the domain of the variable f and Boolean expressions associated with each value from $dom(f)$. It is done according to the following principle. Let us denote by

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

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

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

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

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

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

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

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

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

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

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

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

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

In the following assume that the graph of strongly connected components G_D^{SCC} enables to:

1. Get the set of dependent variables W^* of the variable f via the function `GetDependentVariables(`G_D^{SCC}`,` f`)`.

2. Get the set of variables V^* belonging to the same strongly connected component as f via the function `GetVariablesFromTheSameStronglyConnectedComponent(`G_D^{SCC}`, f)`.
3. Get the instruction A^* to calculate the value of variables from the same strongly connected component as f via the function `GetInstructionToCalculate(`G_D^{SCC}`, f)`.
4. Get the default variable assignment ι^* of variables from the same strongly connected component as f via the function `GetDefaultAssignment(`G_D^{SCC}`, f)`.

Then the algorithm to calculate the domains of variables and the associated Boolean expressions works as follows:

```
ComputeVariableDomain(f, G_D^SCC, C, Φ, D) returns ⟨f, dom(f)⟩
  if ⟨f, dom(f)⟩ ∈ D then return ⟨f, dom(f)⟩
  else
    W* ← GetDependentVariables(G_D^SCC, f);
    V* ← GetVariablesFromTheSameStronglyConnectedComponent(G_D^SCC, f);
    A* ← GetInstructionToCalculate(G_D^SCC, f);
    ι* ← GetDefaultAssignment(G_D^SCC, f);
    D̄ = ∅
    forall w ∈ W* do
      D̄ ← D̄ ∪ ComputeVariableDomain(w, G_D^SCC, C, Φ, D)
    done
    Σ ← ComputeCartesianProduct(D̄);
    ∀v ∈ V* dom(v) = ∅, ∀v ∈ V* φ_v = ∅
    forall σ ∈ Σ do
      φ_σ = ⋀_{w∈W*} φ_(w,σ(w)), φ_(w,σ(w)) ∈ Φ
      Let τ : W* ∪ V* → C
      ∀v ∈ W* τ(v) = σ(v)
      τ = Propagate(A*, ι*, τ)
      forall v ∈ V* do
        dom(v) ← dom(v) ∪ {τ(v)}
        φ_(v,τ(v)) ← φ_(v,τ(v)) ⋁ φ_σ
        φ_v ← φ_v ∪ {φ_(v,τ(v))}
      done
    done
    forall v ∈ V* do
      D ← D ∪ ⟨v, dom(v)⟩
      Φ ← Φ ∪ φ_v
      C ← C \ {v}
      if (v == f) then p = ⟨v, dom(v)⟩
    done
    return p
```

Remark 4.2. *In the case of a Data-Flow assertion, the dependency graph does not contain any cycles and the number of strongly connected components is equal to the number of variables and also to the number of the assignments (as long as each variable is assigned only once). Since the number of variables used in the right hand side of the assignments is never big, this operation is performed very efficiently.*

Example 4.6 (A Data Gathering and Processing Network)**.** Let us explain how this algorithm works for the independent assertion of the red wire example of this chapter. The Boolean formulae for variables of the independent GTS of the Network system $W_i.s, i = 1..3$, $W_i.outFlow, i = 1..3$,

$SW_i.s, i = 1..6$, $SW_i.inFlow, i = 1, 3$ and $P_i.s, i = 1, 2$ have been calculated by the previous step of the algorithm. The variables of the independent assertion V^* (i.e. the flow variables) depend on them via the assertion A^*. The dependency graph of A^* contains cycles, because acausal components have been used in the model. Strongly connected components of this dependency graph are represented Figure 4.9.

Variables $SW_i.leftFlow$, $SW_i.rightFlow$, $i = 1..6$ belong to the same strongly connected component and depend on variables $SW_i.inFlow, i = 1..6$ and $SW_i.s, i = 1..6$ (see Figure 4.9). Let us denote this set of variables V_0^*. The Boolean equations for the variables from the same strongly connected component are calculated together.

1. We start with
$$\phi_{(v,c)} = \emptyset, \forall v \in V_0^*, c \in dom(v)$$
2. For each assignment σ of variables $SW_i.inFlow, i = 1, 3$, $SW_i.s, i = 1..6$
   (a) First, the corresponding Boolean formula ϕ_π is calculated.
   (b) Then, the values of the variables from V_0^* are calculated according to the assertion A^* and the assignment σ.
   (c) Finally, for each variable $v \in V_0^*$ and its value c, the corresponding Boolean formula $\phi_{(v,c)}$ is updated as follows:
$$\phi_{(v,c)} \leftarrow \phi_{(v,c)} \bigvee \phi_\sigma$$

For example, consider a configuration where W1, W3 and SW6 are failed and all the other components are working. In that case $SW1.inFlow = true$ and $SW3.inFlow = false$. The corresponding Boolean equation is as follows:

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

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

$$\phi_{(SW1.leftFlow,true)} \leftarrow \phi_{(SW1.leftFlow,true)} \bigvee \phi_\sigma$$
$$\vdots$$
$$\phi_{(SW6.leftFlow,true)} \leftarrow \phi_{(SW6.leftFlow,true)} \bigvee \phi_\sigma$$

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

□

4.3.6 Results

In this section, we give some experimental results for the red wire example of this chapter obtained by compilation of the model into Fault Trees. For technical reasons, the Fault Trees generated by the AltaRica 3.0 compiler are quite different from those an analyst would write. The minimal cutsets are however the expected ones. For instance, the minimal cutsets, their probabilities and contributions for the top event *"P_1 cannot send data to the plant"*, defined by the observer `P1failed`, are given in Table 4.3. The same results for the top events *"P_2 cannot send data to the plant"* (observer `P2failed`)

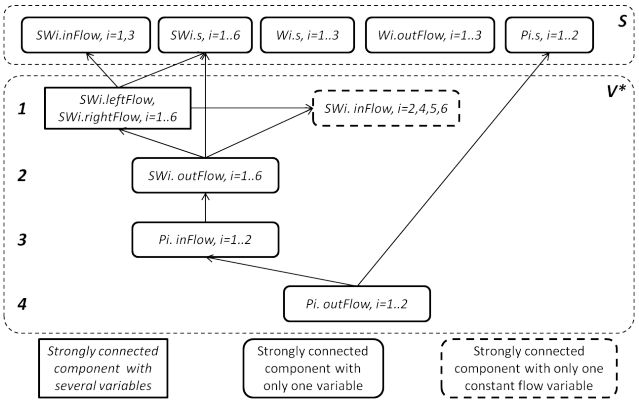


Figure 4.9: Dependency graph of the Independent Assertion

and *"Neither P_1 nor P_2 can send data to the plant"* (observer `P1P2failed`) are summarized in Table 4.4 and in Table 4.5 respectively.

Note that the top events are specified via the observers (e.g. observers `P1failed`, `P2failed`, `P1P2failed`) and their value (e.g. `true`). Several observers can be defined for the same AltaRica 3.0 model. Thus, several Fault Trees can be generated from a unique model.

From the generated Fault Trees, it is also possible to calculate probabilities of the top events, importance factors, etc. Probabilities of the top events, calculated for the period from 0 to 1000 hours, are presented in Figure 4.10.

The generated Fault Trees have been assessed with XFTA [1] calculation engine.

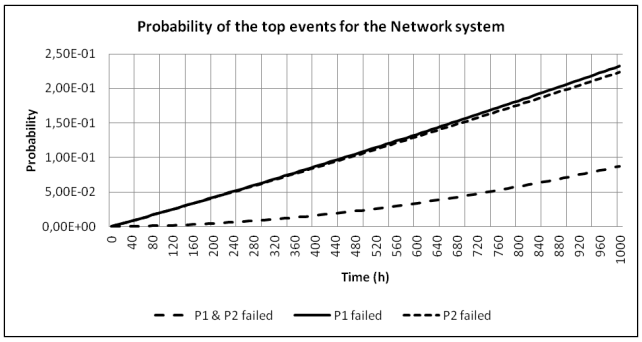


Figure 4.10: Probability of the top events

The initial model of the Data Gathering and Processing network contains 45 variables and 12 events. The complete reachability graph contains 1536 states and 9216 transitions. The partitioned model has 9 parts: reachability graphs of 8 parts are very similar and contain only 2 states, the last one contains 6 states (see Figure 4.7). For comparison, execution times of the program with and without partitioning for the red wire example of this article are given in Table 4.2. These results have been obtained on a laptop computer with a single processor Intel Core i7, a 6GB memory and running on Windows 7.

Table 4.2: Execution times of the program for the model of the Network system

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

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

Table 4.3: Minimal cutsets for the top event "P1 cannot send data to the plant"

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

Table 4.4: Minimal cutsets for the top event "P2 cannot send data to the plant"

Rank	Order	Probability	Contribution	Minimal cutset
1	2	0.00905592	0.103344	P1.failure P2.failure
2	2	0.00905592	0.103344	P1.failure SW4.failure
3	2	0.00905592	0.103344	P2.failure SW5.failure
4	2	0.00905592	0.103344	SW4.failure SW6.failure
5	2	0.00905592	0.103344	SW1.failure SW4.failure
6	2	0.00905592	0.103344	SW4.failure SW5.failure
7	2	0.00905592	0.103344	SW3.failure SW5.failure
8	2	0.00905592	0.103344	SW1.failure SW3.failure
9	2	0.00905592	0.103344	SW3.failure SW6.failure
10	1	0.00498752	0.0569162	ccf
11	2	0.000946884	0.0108056	SW1.failure W3.failure
12	3	9.01079e-5	0.00102829	SW2.failure SW5.failure W3.failure
13	3	9.01079e-5	0.00102829	SW2.failure SW6.failure W3.failure
14	3	9.42165e-6	0.00010752	SW3.failure W1_failure W2.failure
15	3	9.85124e-7	1.1242e-5	W1_failure W2.failure W3.failure
16	4	8.96588e-7	1.02316e-5	SW2.failure SW4.failure W1_failure W2.failure

Table 4.5: Minimal cutsets for the top event "Neither P1 nor P2 can send data to the plant"

4.4 Compilation of stochastic models

Now consider a stochastic Guarded Transition System

$$G = \langle V, E, T, A, \iota, delay, expectation \rangle,$$

as defined in Section 2.7. In that case, a delay of firing $d(t)$ is associated with each transition $t \in T$. It is calculated according to the rules defined in Section 2.7. To compile Timed/Stochastic GTS into Fault Trees or critical sequences of events, we need to abstract from the delays of transition firings. We consider only two possible cases:

- immediate transitions, and
- timed stochastic transitions.

For all immediate transitions, the delay of firing is set to 0; for all other transitions it is set to ∞. Let σ be an assignment of variables from V, then the delay of transition firing $d(t)$ is abstracted as follows:
$\forall t = \langle e, G, Q \rangle \in T$, such that $\sigma(G) = true$

$$d(t) = \begin{cases} 0 & \text{if } delay(e) = 0 \\ \infty & \text{otherwise.} \end{cases}$$

The corresponding reachability graph $\Gamma = (\Sigma, \Theta)$ is defined as follows:

1. The initial state $\sigma_0 = Propagate(A, \iota, \iota)$ belongs to Σ.

2. If $\sigma \in \Sigma$ and there is a transition $t = \langle e, G, Q\rangle \in T$ with $d(t) = 0$, then the state $\tau = Fire(Q, A, \iota, \sigma)$ belongs to Σ and the transition (σ, e, τ) belongs to Θ.

3. If $\sigma \in \Sigma$ and there is a transition $t = \langle e, G, Q\rangle \in T$ with $\sigma(G) = true$ and $d(t) = \infty$ and there is no transitions $t' \in T$, such that $d(t') = 0$, then the state $\tau = Fire(Q, A, \iota, \sigma)$ belongs to Σ and the transition (σ, e, τ) belongs to Θ.

Note that in the case of immediate transitions (case 2), the probability $p(t_i)$ (i.e. the probability to be fired) associated with each immediate transition $t_i = \langle e_i, G_i, P_i\rangle$ fireable in state σ is calculated as follows:

$$p(t_i = \langle e_i, G_i, P_i\rangle) = \frac{expectation(e_i)}{\sum\limits_{t_k = \langle e_k, G_k, P_k\rangle : d(t_k) = 0} expectation(e_k)}$$

Step 2 of the compilation algorithm is slightly modified to take into account the definition of the reachability graph given above. The reachability graph is constructed according to the algorithm given Figure 4.11.

```
C ← {⟨σ0, 0⃗⟩}, D ← ∅
while C ≠ ∅ do
  Let ⟨σ, P⃗⟩ ∈ C
  flag ← false
  C ← C \ {⟨σ, P⃗⟩}, D ← D ∪ {⟨σ, P⃗⟩}
  forall t = ⟨e, G, Q⟩ ∈ T, t) = 0 do
    flag ← true
    τ = Fire(Q, A, ι, σ)
    C ← C ∪ {⟨τ, P⃗[e] ← 1⟩}
  done
  if not flag then
    forall t = ⟨e, G, Q⟩ ∈ T, σ(G) = true do
      τ = Fire(Q, A, ι, σ)
      C ← C ∪ {⟨τ, P⃗[e] ← 1⟩}
    done
  flag ← false
done
```

Figure 4.11: The algorithm to compile a GTS into Boolean expressions

Example 4.7 (Spare workstations with on demand failures)**.** To illustrate how the algorithm works, consider the subsystem of the Data Gathering and Processing network from the Example 4.1, composed of two spare workstations W1 and W2. Moreover assume that when the spare workstation W2 is attempted to start, it fails on demand with a probability γ, and is correctly started with a probability $1 - \gamma$.

The GTS representing a spare workstation with on demand failures is depicted Figure 4.12 and the corresponding AltaRica 3.0 code is as follows:

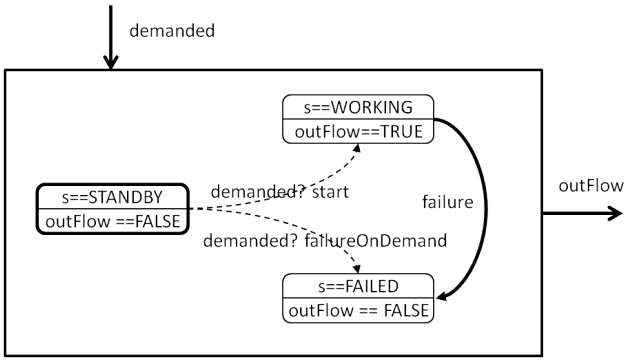


Figure 4.12: GTS of a spare workstation with on demand failures

```
domain SpareComponentState {STANDBY, WORKING, FAILED}
class OnDemandWorkstation
  SpareComponentState s (init = STANDBY);
  Boolean outFlow (reset = false);
  Boolean demanded (reset = false);
  event start(delay = 0, expectation = 1 - gamma);
  event failureOnDemand(delay = 0, expectation = gamma);
  event failure(delay = exponential(lambda));
  parameter Real lambda = 1.0e-5;
  parameter Real gamma = 0.01;
transition
  start: s == STANDBY and demanded -> s := WORKING;
  failureOnDemand: s == STANDBY and demanded -> s := FAILED;
  failure: s == WORKING -> s := FAILED;
assertion
  outFlow := s == WORKING;
end
```

The AltaRica 3.0 model of the subsystem is given below:

```
block TwoSpareWorkstations
  OnDemandWorkstation W1(s.init = WORKING);
  OnDemandWorkstation W2(s.init = STANDBY);
  Boolean outFlow (reset = false);
assertion
  W1.demanded := true;
  W2.demanded := W1.s == FAILED;
  outFlow := W1.outFlow or W2.outFlow;
end
```

In the initial state the workstation W1 is working and the workstation W2 is in standby mode. When the workstation W1 is failed, the flow variable `W2.demanded` becomes true and the immediate transitions `W2.start` and `W2.failureOnDemand` become fireable. The corresponding reachability graph is depicted Figure 4.13. Immediate transitions are represented by dashed lines and timed transitions by plane lines. The initial state $S0$ is marked in bold. The failure state is $S2$ (it is marked in grey). It corresponds to the state, where both workstations W1 and W2 are failed.

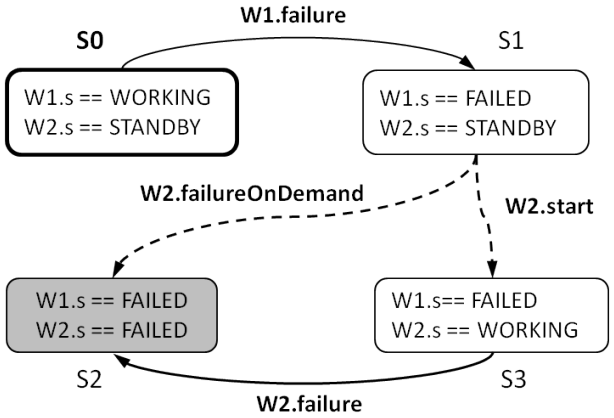


Figure 4.13: Reachability graph of the system made of two spare workstations

This reachability graph is then transformed into a set of Boolean equations. Boolean equation corresponding to the failure state $S2$ is as follows:

$$\phi_{S2} = W1.failure \bigwedge W2.failureOnDemand \bigwedge \overline{W2.failure} \bigwedge \overline{W2.start} \bigvee$$
$$\bigvee W1.failure \bigwedge W2.start \bigwedge W2.failure \bigwedge \overline{W1.failureOnDemand}$$

Note that the sequencing between events is lost during the compilation. For example, the sequence **W1.failure, W2.failureOnDemand** is transformed into the conjunction

$$W1.failure \bigwedge W2.failureOnDemand.$$

The minimal cutsets of this Fault Tree are:

- $\{W1.failure, W2.failureOnDemand\}$
- $\{W1.failure, W2.failure, W2.start\}$

In the case of the compilation into critical sequences of events, the following sequences are generated: $\phi_{S2} = W1.failure \cdot W2.failureOnDemand + W1.failure \cdot W2.start \cdot W2.failure$

4.5 Complexity Analysis and correctness

4.5.1 Complexity

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

Step 1: *Partitioning* is a syntactical operation. The complexity of this step is linear on the size of the GTS model generated by the previous step.

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

Step 3: *Compilation of the reachability graphs into Boolean formulae* is linear on the size of the graph.

Step 4: The independent assertion is compiled symbolically into Boolean equations. The complexity of this operation depends on the number of strongly connected components of the dependency graph of A^*. Let us consider that the dependency graph has m strongly connected components. Note that in case of a Data-Flow assertion m is equal to the number of variables in V^*. The complexity of each step of the algorithm is exponential on the number of outgoing edges of each strongly connected component. But in general the number of outgoing edges (i.e. the number of variables used in the right hand side of the assignments and in conditions) is never big and then can be considered as constant. Thus, the complexity can be considered as linear on the number of strongly connected components of the dependency graph of A^*. In the best case, when the assertion A^* is Data-Flow, the complexity is linear on the number of flow variables assigned in A^* ($O(m)$). In the worst case, there is only one strongly connected component ($m = 1$) and there are n outgoing edges, where n is the number of state variables of the model, the complexity is exponential on n and the advantage of the partitioning is lost.

4.5.2 Correctness

The correctness of the algorithm relies on the following properties.

Property 4.1. *Let $G = G_1 \times G_2 \times \ldots \times G_n \uplus < V^*, A^*, \iota^* >$ be a partitioned GTS and let $\Gamma = (\Sigma, \Theta)$ be a reachability graph of G. Then*

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

where

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

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

1. *$\forall v \in V \setminus V^*$ $\tau(v) = \sigma(v)$*
2. *$\tau = Propagate(A^*, \iota^*, \tau)$*

Proof 4.1. *The proof is based on the fact that the GTS G_i are built over distinct sets of variables and transitions and on the fact that the variables from V^* are also distinct from the variables of the independent Guarded Transtion Systems.* □

Let $G = \langle V, E, T, A, \iota \rangle$ be a partitioned GTS:

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

Let $\Gamma = (\Sigma, \Theta)$ be its reachability graph. Let $\Gamma_i = (\Sigma_i, \Theta_i)$ be the reachability graph of G_i $\forall i = 1..n$. Let $\phi_{(v,c)}, v \in V, c \in dom(v)$ be a Boolean equation generated from G by the algorithm described in Section 4.3.
According to this algorithm:

$$\phi_{(v,c)} = \bigvee_{\sigma \in \Sigma : \sigma(v) = c} \phi_\sigma, \tag{4.1}$$

where ϕ_σ is a Boolean expression associated with the state $\sigma \in \Sigma$.

Since $\Gamma = (\Gamma_1 \otimes \Gamma_2 \otimes \ldots \otimes \Gamma_n)|_{<V^*,A^*,\iota^*>}$, each state σ of Γ can be represented as follows:

$$\sigma = (\sigma_1 \circ \sigma_2 \circ \ldots \circ \sigma_n)|_{<V^*,A^*,\iota^*>}, \sigma_i \in \Sigma_i \forall i = 1..n \tag{4.2}$$

Then the Boolean expression associated with the state $\sigma \in \Sigma$ can be expressed as a conjunction of Boolean expressions $\phi_{\sigma_i}, \sigma_i \in \Sigma_i$ associated with states of independent reachability graphs Γ_i:

$$\phi_\sigma = \bigwedge_{i=1..n} \phi_{\sigma_i}, \sigma_i \in \Sigma_i \tag{4.3}$$

As explained in section 4.3.4, ϕ_{σ_i} are obtained by the compilation of the paths of the independent reachability graphs Γ_i.

Property 4.2. *Let $G = \langle V, E, T, A, \iota\rangle$ be a partitioned GTS:*

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

Let $\Gamma = (\Sigma, \Theta)$ be its reachability graph. Let $\Gamma_i = (\Sigma_i, \Theta_i)$ be the reachability graph of G_i, $\forall\ i = 1..n$. Let $\phi_{(v,c)}$ be a Boolean expression, generated by the algorithm. If $\pi = e_1, e_2, \ldots, e_k$, $e_j \in E\ \forall i = 1..k$, is a path in the reachability graph Γ, such that $(\sigma_0, e_1, \sigma_1) \in \Theta, (\sigma_1, e_2, \sigma_2) \in \Theta, \ldots, (\sigma_{k-1}, e_k, \sigma_k) \in \Theta$ and $\sigma_k(v) = c$, then $e_1 \wedge \ldots \wedge e_k$ is a cutset of $\phi_{(v,c)}$.

Proof 4.2. *Since $\Gamma = (\Gamma_1 \otimes \Gamma_2 \otimes \ldots \otimes \Gamma_n)|_{<V^*,A^*,\iota^*>}$, we can write that*

$$\sigma_0 = (\sigma_0^1 \circ \sigma_0^2 \circ \ldots \circ \sigma_0^n)|_{<V^*,A^*,\iota^*>}, \sigma_0^i \in \Sigma_i\ \forall\ i = 1..n$$

and

$$\sigma_k = (\sigma_k^1 \circ \sigma_k^2 \circ \ldots \circ \sigma_k^n)|_{<V^*,A^*,\iota^*>}, \sigma_k^i \in \Sigma_i\ \forall\ i = 1..n$$

The path π can be projected on the independent reachability graphs Γ_i and represents the path from σ_0^i to σ_k^i. This path can be empty. Let us denote by π_{Γ_i} the projection of the path π on the reachability graph Γ_i. According to the rules of compilation of reachability graphs defined in Section 4.3.4, the Boolean expression $\phi_{\sigma_k^i}$ associated with the state σ_k^i is as follows:

$$\phi_{\sigma_k^i} = \bigwedge_{e_k \in \pi_{\Gamma_i}} e_k \bigwedge_{e_j \in E_i, e_j \notin \pi_{\Gamma_i}} \overline{e_j} \bigvee \phi',$$

where ϕ' is a Boolean expression built from other paths of the graph (it may be empty).

Then from 4.3

$$\phi_{\sigma_k} = \bigwedge_{i=1..n} \phi_{\sigma_k^i}$$

$$\phi_{\sigma_k} = \bigwedge_{i=1..n} \Big(\bigwedge_{e_k \in \pi_{\Gamma_i}} e_k \bigwedge_{e_j \in E_i, e_j \notin \pi_{\Gamma_i}} \overline{e_j} \bigvee \phi' \Big)$$

$$\phi_{\sigma_k} = \bigwedge_{e_k \in \pi} e_k \Big(\bigwedge_{i=1..n} \Big(\bigwedge_{e_j \in E_i, e_j \notin \pi_{\Gamma_i}} \overline{e_j}) \Big) \Big) \bigvee \phi'',$$

where ϕ'' represents other terms of the disjunction. As a consequence

$$\bigwedge_{e_k \in \pi} e_k$$

is a cutset of ϕ_{σ_k}.

From 4.1 it is also a cutset of $\phi_{(v,c)}$. □

Property 4.3. *Let $G = \langle V, E, T, A, \iota \rangle$ be a partitioned GTS:*

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

Let $\Gamma = (\Sigma, \Theta)$ be its reachability graph. Let $\Gamma_i = (\Sigma_i, \Theta_i)$ be the reachability graph of G_i $\forall i = 1..n$. Moreover assume that all the paths in the reachability graphs are labeled with distinct events (assuming that events represent only failures of components, that means that a component cannot be failed twice). Let $\phi_{(v,c)}$ be a Boolean expression, generated by the algorithm. If $e_1, e_2, \ldots, e_k$, $e_j \in E$ $\forall$ $i = 1..k$, is a cutset of $\phi_{(v,c)}$, then there is a path in the reachability graph Γ from σ_0 to σ, such that $\sigma(v) = c$ labeled by the events from the cutset.

Proof 4.3. *If $e_1 \wedge e_2 \wedge \ldots \wedge e_k \models \phi_{(v,c)}$ then from Equation 4.1*

$\exists$ at least one state $\sigma \in \Sigma$ such that $e_1 \wedge e_2 \wedge \ldots \wedge e_k \models \phi_\sigma$.

From Equation 4.3 $\phi_\sigma = \bigwedge_{i=1..n} \phi_{\sigma_i}, \sigma_i \in \Sigma_i$ and as a consequence the projection of the conjunction $e_1 \wedge e_2 \wedge \ldots \wedge e_k$ to each Γ_i satisfies ϕ_{σ_i}:

$$\forall i = 1..n \; e_1 \wedge e_2 \wedge \ldots \wedge e_k|_{\Gamma_i} \models \phi_{\sigma_i}$$

From now, we must prove that the projection of the cutset into Γ_i corresponds to a path in the reachability graph Γ_i. As we have assumed that all the events in a path are distinct, then there is a correspondence (the order does not matter) between the paths in the graph and the cutsets generated from the paths. So, by compilation of independent reachability graphs (see section 4.3.4), the projection corresponds to a path from σ_i^0 to σ_i (this path may be empty).

The local paths in Γ_i are combined into a set of corresponding paths in the reachability graph Γ. □

Remark 4.3. *Assume that the events can be repeated along the paths in the graphs. This may happen when we try to compile models with repairs, i.e. models whose graphs contain cycles. It also may happen when there is a transition of reconfiguration which is fired after a set of failures. In that case, information is lost during the compilation of the path into a conjunction of events. If $e_1, e_2, \ldots, e_n$ is a cutset, it corresponds, in fact, to a set of words built over the alphabet $\{e_1, e_2, \ldots, e_n\}$, such that each letter (event) e_i appears at least once in the word. At least one of these words corresponds to the path in the graph.*

Summary

In this chapter we presented the algorithm to compile Guarded Transition Systems into Fault Trees and critical sequences of events. This algorithm uses advanced partitioning techniques to take advantage from the independence of components. It is efficient on models containing independent parts. It looses its efficiency if the model contains only one partition (and is quite big) or if the assertion cannot be divided into independent parts (in that case the advantage of the partitioning is lost).

In the next chapter we will present the overall architecture of the Modeling, Simulation and Assessment platform of AltaRica 3.0 developed as a part of this thesis.

Chapter 5

AltaRica 3.0 Modeling, Simulation and Assessment Platform

This PhD thesis is done as a part of the AltaRica 3.0 project which aims to develop a Modeling, Simulation and Assessment platform for AltaRica 3.0. The goal of this chapter is to present the architecture of this platform and to describe the developed prototypes. Section 5.1 discusses the motivations of the project. Section 5.2 describes the overall architecture of the platform. Section 5.3 presents the core library of the platform. Section 5.4 gives an overview of the implementation of the stepwise simulator. Section 5.5 describes the implementation of the compiler of AltaRica 3.0 models into Guarded Transition Systems (GTS). Finally, section 5.6 presents the implementation of the compiler of GTS into Fault Trees.

5.1 Motivations: the AltaRica 3.0 project

The aim of the AltaRica 3.0 project [82] is to develop a modeling, simulation and assessment platform to perform Safety Analyses with AltaRica 3.0 modeling language. Figure 5.1 presents the overview of the project.

The new version of the AltaRica language is in the core of this project. It increases the expressive power of AltaRica Data-Flow without decreasing the efficiency of the assessment algorithms. The semantics of AltaRica 3.0 is defined in terms of Guarded Transition Systems (GTS) (see Chapter 2).

The project aims to develop the following assessment tools:

- The compiler from AltaRica 3.0 to Guarded Transition Systems;
- The stepwise simulator for Guarded Transition Systems;
- The graphical simulator of AltaRica 3.0 models;
- The compiler from Guarded Transition Systems to Fault Trees;
- The Fault Tree assessment tool XFTA;
- The Sequence Generator for Guarded Transition Systems;
- The compiler from Guarded Transition Systems to Markov chains;
- XMRK, a tool to assess multi-phase Markov chains with rewards;
- The Stochastic Simulator for Guarded Transition Systems;
- The Model-checker for Guarded Transition Systems;

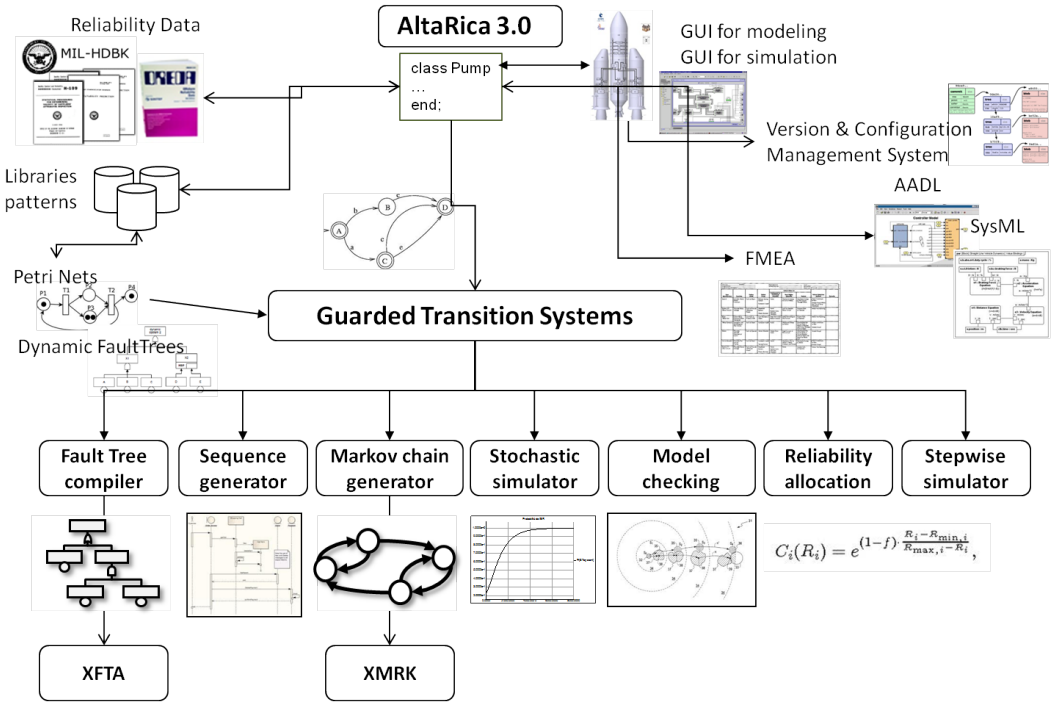


Figure 5.1: The AltaRica 3.0 project

- The Reliability allocation module for Guarded Transition Systems.

These tools enable the user to perform virtual experiments on systems, to perform end-to-end risk assessment with AltaRica 3.0 and also to do cross check calculations. Thanks to these tools AltaRica models can be used to perform Preliminary System Safety Analysis (PSSA) and System Safety Analysis (SSA).

In other words, with AltaRica 3.0 models, it will be possible:

- To perform Fault Tree Analysis (FTA) for static and some kinds of dynamic models;
- To calculate different probabilistic indicators for dynamic models using Markov chain analysis;
- To perform stochastic simulation of dynamic models;
- To verify system and models properties using model-checking techniques;
- To graphically simulate the model in order to validate it;
- To perform reliability and availability allocation for different components given the overall objective.

AltaRica models are first compiled into Guarded Transition Systems (GTS). As seen in Chapter 2, GTS generalize classical safety formalisms, such as Reliability Block Diagrams and Markov chains. It is a pivot formalism for Safety Analyses: other safety models can be compiled into GTS to take benefits from the assessment tools.

In the AltaRica 3.0 project, it is also planned to develop bridges with other tools, especially to work on the integration of system architecture with Safety Analyses through the development of methods and tools to synchronize models of both disciplines.

As part of my thesis, the prototypes of the following tools have been developed:

- The compiler from AltaRica 3.0 into Guarded Transition Systems;
- The compiler from Guarded Transition Systems into Fault Trees;
- The stepwise simulator for Guarded Transition Systems;
- The AltaRica 3.0 textual editor based on Eclipse XText.

Moreover, the overall architecture of the platform has been defined. The achieved developments are described in the following sections.

5.2 Overall architecture of the platform

The overall architecture of the platform is given Figure 5.2. All prototypes of the platform have been developed in C++ to ensure their efficiency.

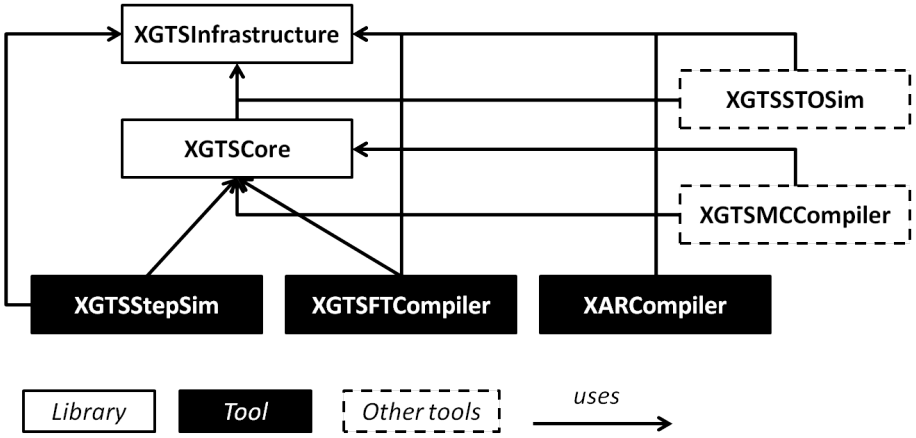


Figure 5.2: Architecture of the platform

XGTSInfrastructure: Library of basic classes. It includes containers, graph structures, parsers for XML files, classes to represent Abstract Syntax Trees (AST), etc... This library is used by the library XGTSCore and all the other tools of the project.

XGTSCore (LOC[1] = 9088): Library of classes, common to all the tools of the project. It contains data structures to represent Guarded Transition Systems, classes to read and write, to optimize, to simulate Guarded Transition System and to generate Reachability graphs for them. It uses the XGTSInfrastructure library and is used by all the other tools of the project.

XARCompiler (LOC = 7584): The compiler from AltaRica 3.0 to Guarded Transition Systems. This tool takes a file containing an AltaRica 3.0 model in input and generates a Guarded Transition System in XML or textual format. It uses both the XGTSInfrastructure and the XGTSCore libraries.

XGTSStepSim (LOC = 985): The stepwise simulator for Guarded Transition Systems. It is an interactive tool. It takes a GTS in XML format in input and enables the user to simulate the model step by step. It uses both the XGTSInfrastructure and the XGTSCore libraries.

XGTSFTCompiler (LOC = 5881): The compiler from Guarded Transition Systems to Fault Trees. It takes a Guarded Transition System in XML format in input and generates the corresponding Fault Tree in Open-PSA format [51]. It uses both the XGTSInfrastructure and the XGTSCore libraries.

XGTSMCGenerator: The compiler from Guarded Transition Systems to Markov chains (using approximation techniques sketched in [21]) It uses both the XGTSInfrastructure and the XGTSCore libraries and is developed by another member of the project.

XGTSSSTOSim The stochastic simulator [9] for Guarded Transition Systems. It uses both the XGTSInfrastructure and the XGTSCore libraries and is based on the compilation techniques developed by M.T. Khuu in his PhD thesis [58]. It is developed by another member of the project.

Each package is compiled via a makefile. All the makefiles have the same structure. They contain the following targets:

install: to create all the necessary directories for the compilation;

compile: to compile the project;

all: to generate all the necessary files (e.g. files generated by flex and bison) and to compile the project;

test: to launch the application on a test case (not available for libraries).

tests: to launch a set of unit tests (not available for libraries).

clean: to clean the project;

cleanAll: to delete all the automatically generated files and to clean the project.

5.3 XGTSCore library

The library XGTSCore contains **9088** lines of code and **55** classes. This library is shared by all the assessment tools. It is composed of the following packages:

XGTSModel: Data structures to represent Guarded Transition Systems (GTS). Partial class diagrams representing GTS are given in Figures 5.3– 5.5.

XGTSXMLParser: To read files with GTS in XML format and load them in memory.

XGTSPrinter: To print a GTS in XML or text format.

XGTSOptimizer: To optimize a GTS according to the principles presented below.

XGTSReachabilityGraph: To generate a reachability graph of a GTS according to the definition given in Section 2.5.2.

XGTSStepSimulator: To simulate a GTS step by step. The stepwise simulation is performed according to the algorithm described in Section 2.6.3.

5.3.1 Optimization of Guarded Transition Systems

The optimization of a Guarded Transition System is done in two steps. The first step consists in the separation of the assertion into independent parts according to the algorithm described in Section 2.6.3. This optimization enables:

- To detect if the assertion is Data-Flow (see definition 2.17) and to find the optimal execution order of the instructions in the assertion.

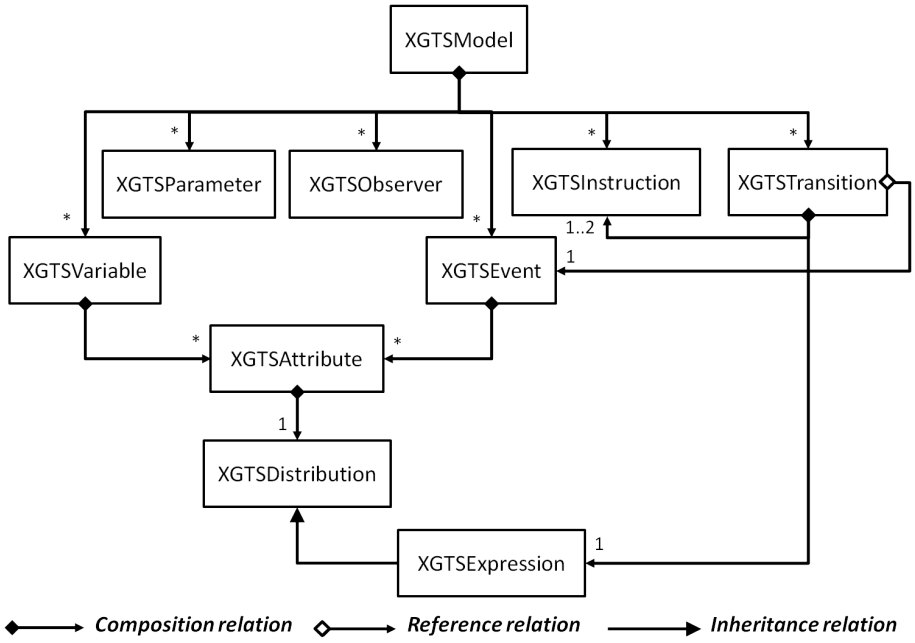


Figure 5.3: GTS class diagram: global view

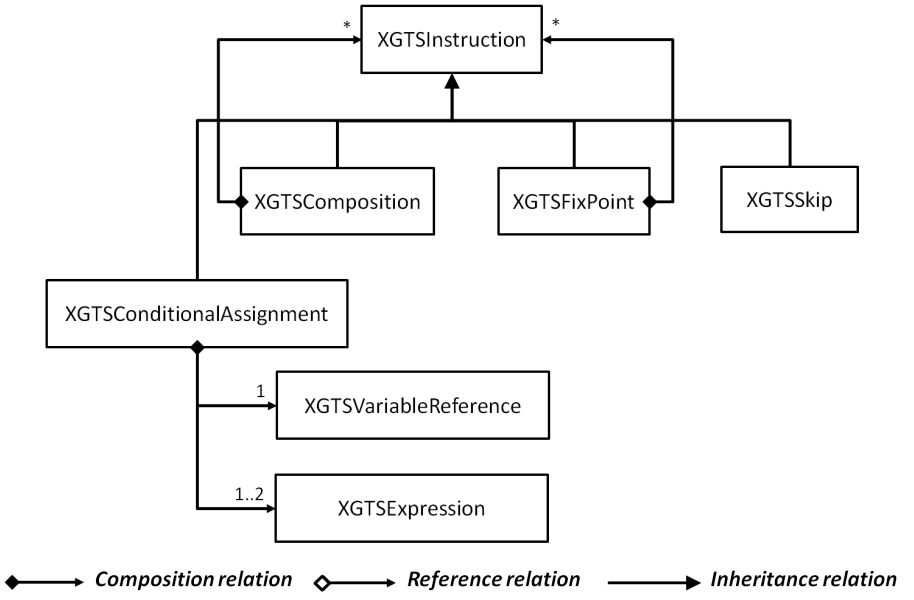


Figure 5.4: GTS class diagram: instructions

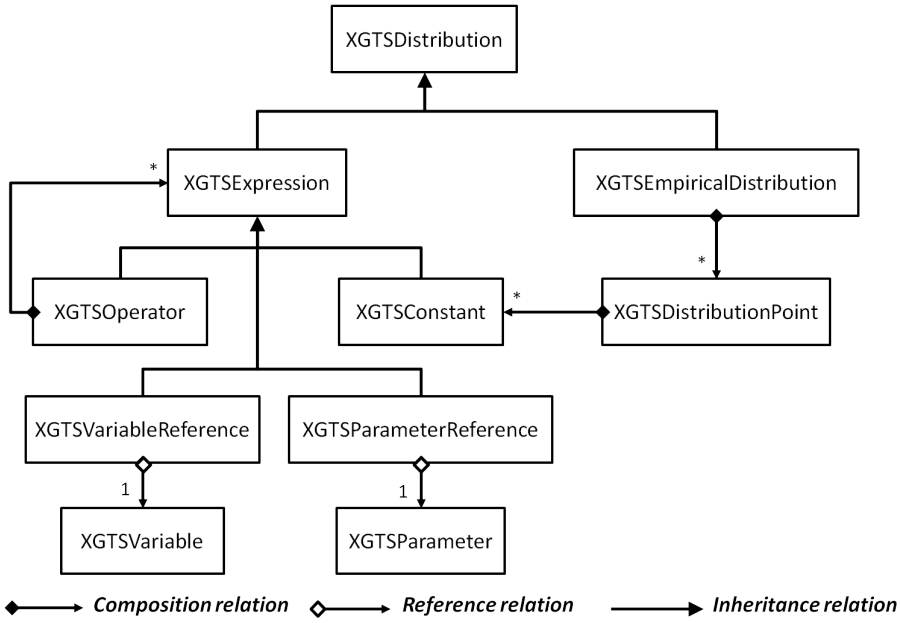


Figure 5.5: GTS class diagram: distributions

- To separate the assertion into independent parts and to find the optimal execution order of the independent parts.

The second optimization is based on the following observation. The firing of a transition is performed in two steps (see Section 2.5):

- First, the action of the transition is executed to update the value of the state variables.
- Second, the assertion is executed to calculate the value of the flow variables according to the value of the state variables.

Since the action of a transition does not generally modify all the state variables of the model, not all the flow variables must be recalculated but only those that depend on the state variables updated by the action of the transition. We use this fact to compute its own assertion for each transition of a model. In practice, this is done using the dependency graph of the assertion (see definition 2.15). This graph enables to find all the flow variables depending on a given set of state variables and to build the instruction to calculate them. Finally, each transition has its own assertion.

The optimization of a GTS can be performed at the compilation of the AltaRica 3.0 model or at the beginning of the assessment by all the tools.

5.4 Stepwise simulator

The stepwise simulator is a command line tool. It enables to load a GTS and to simulate it step by step. The following commands are available:

help: to print the list of available commands.

print gts: to print the loaded GTS model.

print v(ariables)? to print the value of the variables in the current state.

print o(bservers)? to print the value of the observers in the current state.

print tr(ansitions)? to print the list of the transitions, fireable in the current state.

print h(istory)? to print the execution history, i.e. the list of transitions that have been fired from the beginning of the simulation.

fire i to fire the transition number **i**.

back to undo the last transition firing.

reset to restart the simulation.

set (no)? trace to turn on/off the trace mode.

set (no)? display to turn on/off the display mode. In the display mode, the values of the observers and the fireable transitions are automatically displayed after each transition firing.

q(uit)? to exit the program.

To be able to perform graphical simulation of models, it is possible to couple the stepwise simulation with an Interpreter of Graphical Animation models. The principle is described in details in Appendix C.

The stepwise simulator (XGTSStepSim) contains **985** lines of code and **4** classes. It uses the package XGTSStepSimulator of XGTSCore library to execute transitions. The class diagram of XGTSStepSim is depicted in Figure 5.6. The class **XGTSStepSim** is the main class of this project: it performs the execution of user commands listed above. It contains the instances of the following classes:

- **XGTSContext**, which represents the initial context, i.e. the initial value of variables. It is used to reset the simulation.
- **XGTSInterpretedStepper**, which contains an instance of loaded GTS model. It is used to simulate the model.
- **XGTSStepSimMessenger**, which regroups and prints all the messages of the tool. It is a good software engineering practice to regroup all messages of the program together.
- **XGTSStepSimHistoryManager**, which is used to store all the transitions fired by the user from the beginning of the simulation.

5.5 AltaRica 3.0 compiler

AltaRica 3.0 compiler transforms AltaRica 3.0 models (possibly separated into several files) into Guarded Transition Systems according to the principles given in Section 3.5.

The project XARCompiler contains **7584** lines of code and **36** classes. It uses XGTSInfrastructure and XGTSCore libraries. It is composed of the following packages:

XARParser: Contains all the necessary files to automatically generate a parser for AltaRica 3.0 models using *flex* and *bison*.

XARModel: Data structures used to store AltaRica 3.0 models in memory.

XARApplications: Contains all the classes used to compile AltaRica 3.0 models into GTS. They are presented hereafter.

Figure 5.7 represents the different steps of the compilation of an AltaRica 3.0 model:

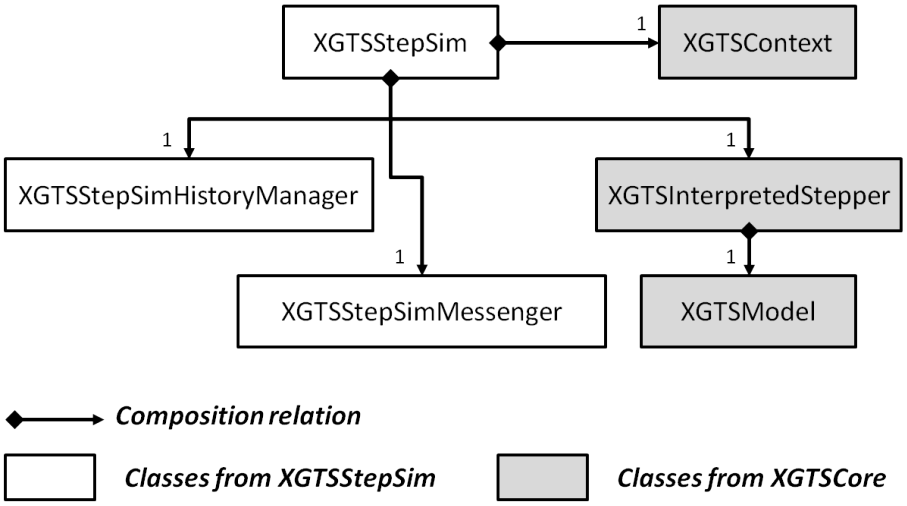


Figure 5.6: Stepwise simulator class diagram

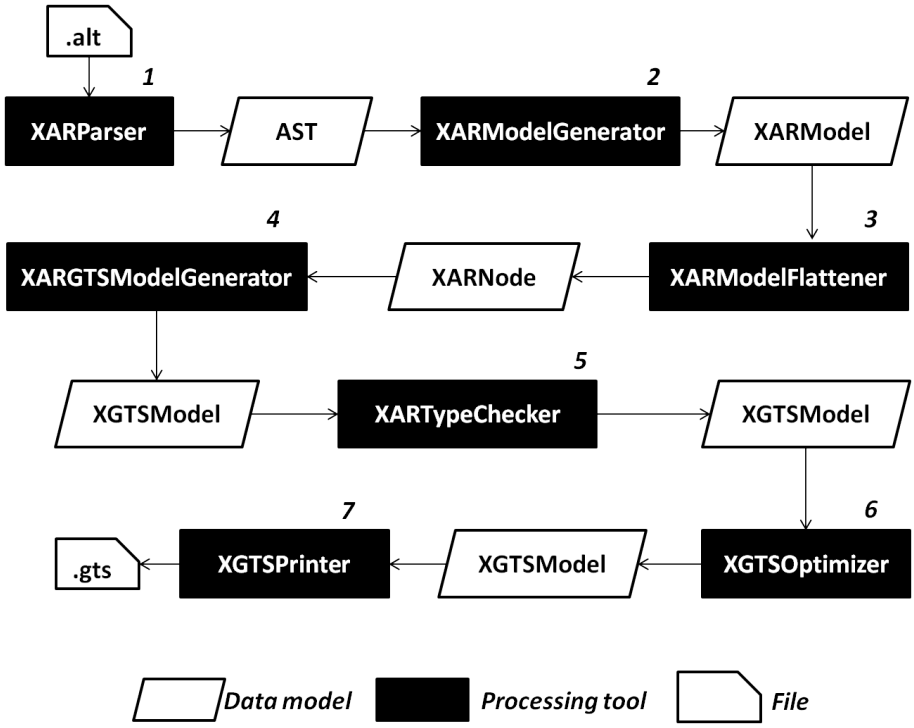


Figure 5.7: Compilation of AltaRica 3.0 models

Step 1 (XARParser): The first step consists in reading an input file (an AltaRica 3.0 model), parsing it and generating a corresponding Abstract Syntax Tree (AST). During this step, the lexical and syntactic rules are verified. XARParser is automatically generated by *flex* and *bison* using a formal grammar description file.

Step 2 (XARModelGenerator): During this step, an AST generated by the previous step is transformed into XARModel, a more appropriate data model to store AltaRica 3.0 models. The class diagram representing an AltaRica 3.0 data model is partially depicted in Figures 5.8 and 5.9.

Step 3 (XARModelFlattener): Then the main block or class (typically the last one in the file or

the one specified by the user) is flattened according to the algorithm described in Section 3.5. During this step, some semantics rules are verified such as the circular definitions of classes and records, the declarations of all used events, classes and blocks, the multiple definitions of declared elements, etc.

Step 4 (XARGTSModelGenerator): During this step, synchronization are flattened according to the algorithm presented in Section 3.5. A Guarded Transition System (XGTSModel) is generated. All remaining semantics rules are verified: the circular definitions in synchronizations, the declaration of all the variables used in the expressions, the declarations of all the events used in the synchronizations, etc.

Step 5 (XARTypeChecker): XARTypeChecker[2] checks types in the instructions and expressions.

Step 6 (XGTSOptimizer): This step is optional. XGTSOptimizer performs GTS optimizations according to the principles given in Section 5.3.1.

Step 7 (XGTSPrinter): During this step, the generated XGTSModel is printed to the output file. It can be printed either in XML format or in text format. Text format is used for debugging, while XML format is used for the assessment tools.

If an error is detected at any step of the compilation, an exception is raised and nothing is generated. All the error messages of the compiler are managed by the class XARCompilerErrorMessenger. It is implemented using the singleton design pattern [40].

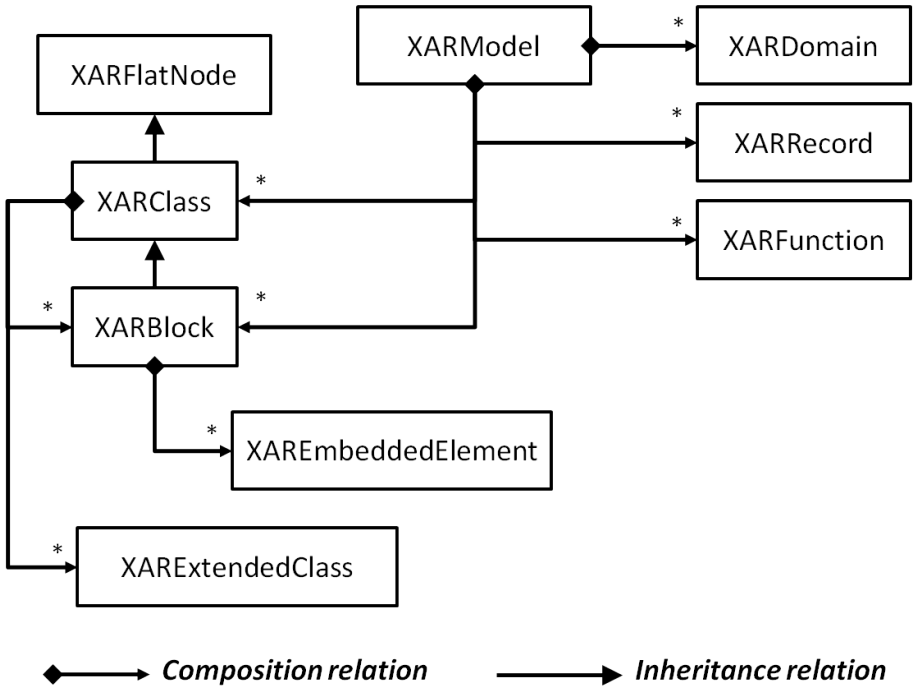


Figure 5.8: AltaRica 3.0 class diagram: part 1

5.6 Fault Tree compiler

The Fault Tree compiler takes a GTS in XML format in input and generates a Fault Tree in Open-PSA format [51] according to the algorithm given in Chapter 4.

[2]Has not yet been implemented in the current version of the prototype.

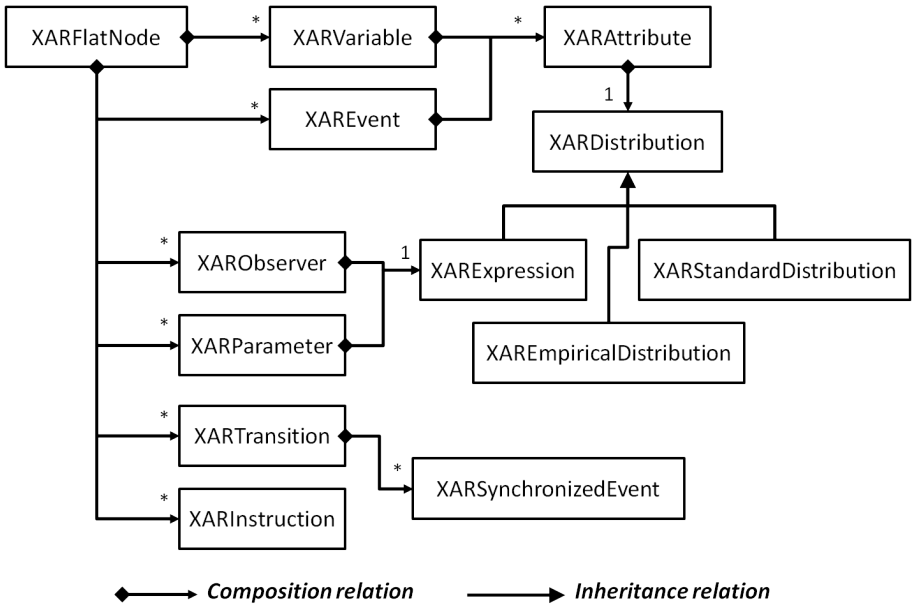


Figure 5.9: AltaRica 3.0 class diagram: part 2

XGTSFTCompiler project contains **5881** lines of code and **28** classes. It is organized in the following packages:

XGTSPartition: Regroups the classes used to partition a given GTS and to store it in memory.

XGTSFTCReachGraph: Regroups the classes used to generate the reachability graph of a given GTS according to the principles explained in Section 4.4.

XGTSBOOLExp: Regroups the classes used to generate the Boolean equations from reachability graphs and from the assertion and to store them in memory.

Figure 5.10 represents the different steps of the compilation of GTS into Fault Trees:

Step 1 (XGTSXMLParser): The first step consists in reading a GTS model in XML format and to load it in the memory to obtain XGTSModel.

Step 2 (XGTSPartitionBuilder): During the second step, the loaded GTS is partitioned according to the algorithm described in Section 4.3.2. A XGTSComposedModel is generated.

Step 3 (XGTSComposedModelOptimizer): The third step consists in optimizing the partitioned GTS. Each independent GTS is optimized according to the principles given in Section 5.3.1. The independent assertion is also optimized according to the algorithm described in Section 2.6.3.

Step 4 (XGTSFTCompiler): The fourth step is detailed in Figure 5.11. For each independent GTS, its reachability graph is generated by XGTSRGBuilder and then this generated reachability graph is compiled into Boolean equations by XGTSDomainBuilder according to the algorithm given in Section 4.3.4. XGTSDomains denotes a set of pairs that associates to each pair (variable, value) its corresponding Boolean equation. Finally, the independent assertion (XGTSAssertion) is compiled into Boolean equations by XGTSDomainBuilder, which generates XGTSDomains according to the algorithm described in Section 4.3.5. The result of this operation is a set XGTSDomains, which regroups all the generated Boolean equations.

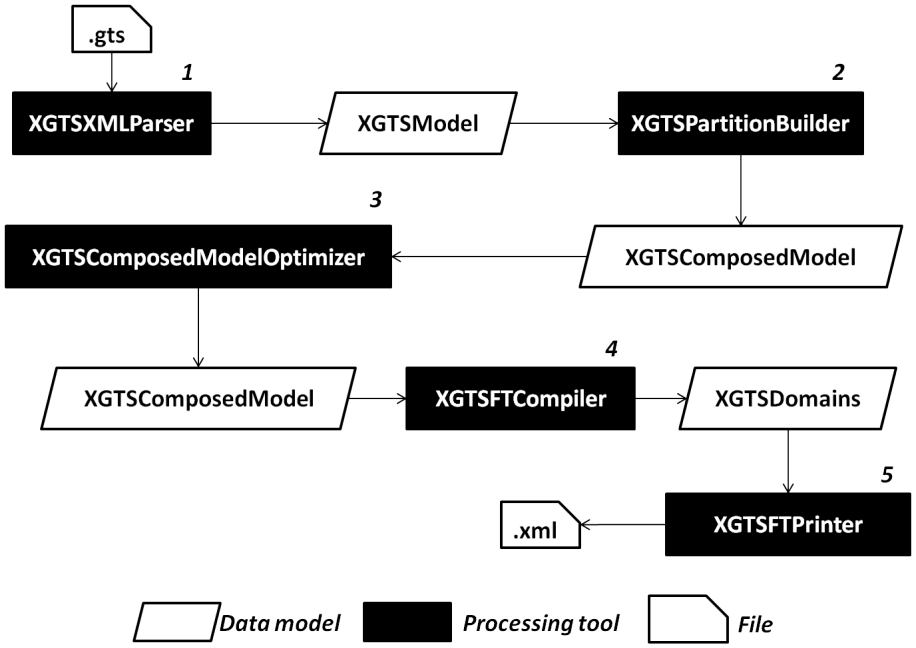


Figure 5.10: Compilation of GTS into Fault Trees

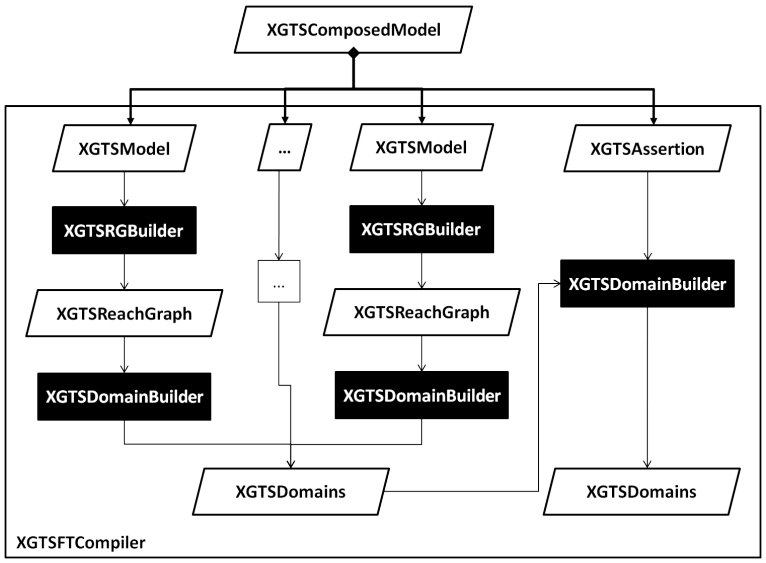


Figure 5.11: Compilation of GTS into Fault Trees: the fourth step

Step 5 (XGTSFTPrinter): During the last step the generated Boolean equations are printed in a file in Open-PSA format.

Figure 5.12 depicts partial the class diagram of the data structures used to store Boolean equations. Alternative (more sophisticated) data structures can be used to store Boolean equations, based on Decision Diagrams [22, 25]. These data structures have not been implemented in the current version of the prototype.

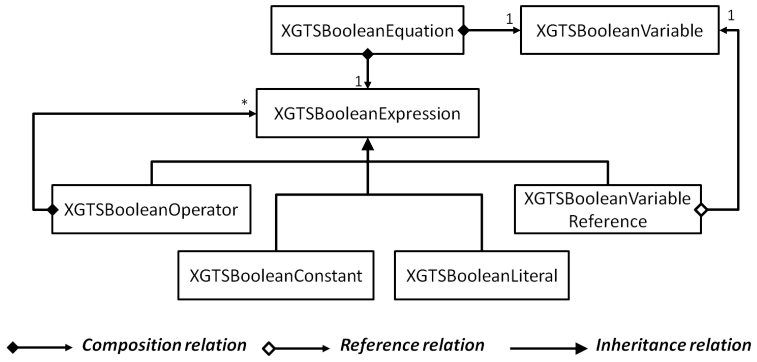


Figure 5.12: Boolean equations: class diagram

Summary

In this chapter, we presented the overall architecture of the Modeling, Simulation and Assessment platform for AltaRica 3.0. The following prototypes have been developed:

- The core library of the project, used to store, to simulate and to process GTS. It is used by all the assessment tools for GTS. Other tools for GTS, e.g. model-checker, can use this library.
- The stepwise simulator of GTS.
- The compiler of AltaRica 3.0 models into GTS.
- The compiler of GTS into Fault Trees.

All the developed prototypes will be the basis of OpenAltaRica project, held by IRT SystemX together with industrial partners such as Airbus, Thales and Safran. The goal of this project is to make high level modeling for Safety Analysis, based on AltaRica 3.0 modeling language, available for a wide audience (both academic and industrial). The main objectives of the project are to:

- Implement tools for editing, animation and assessment of AltaRica 3.0 models;
- Develop libraries of reusable components and modeling patterns;
- Work on the modeling methodologies and training materials.

The developed tools will be improved in order to become robust and efficient to be able to handle industrial scale models.

Chapter 6

Conclusion

In this PhD thesis, we worked on the Model-Based approach for Safety Assessment and, in particular, on the new version of AltaRica – a high level modeling language dedicated to Safety Analysis. Model-Based approach for Safety Assessment presents many advantages compared to classical approaches, such as Fault Tree Analysis.

- First, safety models are kept close to functional and physical architectures of the systems under study. Therefore, it gets much easier to propagate changes in system specifications and to trace changes in safety models.
- Second, models can be graphically animated. The incident or accident scenarios can be visualized and discussed. High level models are much easier to share amongst the different stakeholders than low level models.
- Third, high level modeling favors the reuse of models and knowledge capitalization.
- Fourth, high level modeling languages such as AltaRica have a greater expressive power than Boolean formalisms (e.g. Fault Trees or Reliability Block Diagrams). It is therefore possible to capture phenomena such as spare redundancies, system reconfigurations, etc.
- Fifth, high level modeling languages such as AltaRica provide constructs to structure models into hierarchies of components contrary to flat formalisms such as Markov chains.
- Sixth, high level modeling languages such as AltaRica have a formal semantics defined in terms of state machines. This semantics makes it possible to enlarge the palette of assessment tools with technologies coming from other disciplines, e.g. model-checking.
- Finally, high level modeling languages such as AltaRica make it possible to calculate performance indicators beyond classical reliability indicators. Typically, AltaRica models have been used to assess the average production of plants in presence of hazards (e.g. unavailability of machines or human operators).

This is the reason why many successful industrial experiences using AltaRica (Data-Flow) have been reported. However, even if AltaRica Data-Flow has now reached an industrial maturity, the language and the assessment tools can still be improved. Based on an original viewpoint, which consists in viewing a modeling language as a combination of an underlying mathematical formalism and a paradigm to structure models, the new version AltaRica 3.0, improves the language into two directions.

In the first part of this manuscript, we introduced in details the new underlying mathematical formalism of AltaRica 3.0, the Guarded Transition Systems (GTS). This states/transitions formalism dedicated to Safety Analyses makes it possible to handle systems with instant loops and to define

acausal components (i.e. components for which input and output flows are decided at run time). In addition, Guarded Transition Systems fulfill all the expected requirements of a modeling formalism for Safety Analysis:

- They are event-based.
- They make it possible to easily represent remote interactions by means of flow variables and assertions.
- GTS are compositional. They make it possible to represent reachable states in an implicit way and to structure models into hierarchies of components and subsystems.
- GTS have a versatile synchronization mechanism, which enables to easily represent common cause failures, shared resources, etc.
- GTS make it possible to represent acausal components and to handle looped systems.
- The semantics of (Stochastic) GTS is formally defined.
- Many efficient assessment algorithms are available for GTS.

Indeed, Guarded Transition Systems generalize classical formalisms for Safety Analysis such as Markov chains and Reliability Block Diagrams. They can be seen as a pivot formalism for Safety studies. Other safety models can be compiled into Guarded Transition Systems in order to take advantage from the assessment tools.

Despite all these interesting modeling properties, Guarded Transition Systems still have several limitations. First, they are not designed to capture continuous phenomena, typically modeled with Ordinary Differential Equations. Some formalisms to handle continuous phenomena are Matlab/Simulink, Modelica, Hybrid & Timed Automata. To be able to handle dynamic reliability problems (e.g. when the failure rate of a component depends on its stress), one has to mix continuous and discrete descriptions of the system. Thus, one of the scientific challenges for Guarded Transition Systems will be the introduction of continuous variables in order to handle dynamic reliability problems.

Second, GTS are not designed to handle processes or actors which are dynamically created and destroyed during the mission. Some formalisms to handle dynamically created processes are process Algebras, e.g. π-calculus, Colored Petri-Nets. As seen in appendix A, PEPA nets are more adapted to handle systems with mobile components than GTS. Modeling systems with mobile components and components that can be created and destroyed at run time, is another scientific challenge for Guarded Transition Systems.

In the second part of this manuscript, we presented the new structural constructs of AltaRica 3.0. Safety models usually stand at a rather high abstraction level. Physical details are abstracted away and only important changes of the system are modeled as events. Because they consider systems with a high level of abstraction, safety models naturally emerge from the prototype-oriented paradigm. However, safety models can be reused, for example to take into account the redundancy. Therefore, they also emerge from the object-oriented paradigm. AltaRica 3.0 borrows concepts to both the object-oriented and the prototype-oriented programming:

- *Blocks* can be seen as prototypes and come from prototype-oriented modeling languages,
- *Classes* come from object-oriented modeling languages.

AltaRica 3.0 makes a clear distinction between:

- the stabilized knowledge which is incorporated into libraries of "on-the-shelf" modeling components, for which *classes* are used; and

- the "sandbox" in which the analyst is designing his model of the system under study. In the sandbox, many components are unique (they are represented by *blocks*); some others are instances of reusable components.

In that way, AltaRica 3.0 provides the analyst with powerful structuring constructs that are well-suited for the level of abstraction of Safety Analyses.

Still, other structural operations can be introduced such as the ability to clone *blocks*. The same structural constructs can be used for modeling languages coming from other engineering disciplines (e.g. system architecture). One of the scientific challenges is to show that these structural constructs can be used to synchronize models, coming from different engineering disciplines, and to manage them in the collaborative databases.

In the last part of this manuscript, we worked on the compilation of GTS into Fault Trees and critical sequences of events. The algorithm, proposed in this PhD thesis, extends the algorithm from [88] to the case of Guarded Transition Systems. The first prototype of the Fault Tree compiler has been developed. It has allowed the validation of the algorithm and gives the first good results. This prototype can still be improved by using a more sophisticated data structures, based on Decision Diagrams [22, 25], to handle Boolean equations, in order to be able to support industrial scale models. The generation of critical sequences can also be improved in order to take into account different filtering criteria such the order of the sequence, its probability or its minimality.

This PhD thesis and all the developed prototypes (the common library for GTS, the AltaRica 3.0 compiler, the Fault Tree compiler and the stepwise simulator) will be the basis of the OpenAltaRica project, held by IRT SystemX together with the industrial partners such as Airbus, Thales and Safran. The goal of this project is to make high level modeling for Safety Analysis, based on AltaRica 3.0 modeling language, available for a wide audience (academic and industrial). The main objectives of the project are:

- To implement tools for editing, animation and assessment of AltaRica 3.0 models;
- To develop libraries of reusable components and modeling patterns;
- To work on the modeling methodologies and training materials.

Thanks to this project, AltaRica 3.0 models will be designed for more and more industrial systems. Experience will show how well-adapted the new structural constructs and the underlying mathematical formalism of AltaRica 3.0 are for the design of safety models.

Appendix A

Mobility modeling

In this chapter, we show that AltaRica modeling language can be effectively used to model systems with mobile components and to evaluate their performance and reliability indicators. Many complex systems have mobile components. This is typically the case of the production chains or communication networks. Modeling and performance evaluation of such systems have specific problems. Indeed, modeling formalisms have mostly been designed for systems whose architecture does not change during the mission. Systems with mobile components can be quite naturally described according to the place/component paradigm. In this paradigm, there are two types of objects:

- Topology, i.e. a number of places and neighborhood relations between these places.
- Static or mobile components, each with its own behavior. Any component is situated in a unique place. Mobile components can change their places. Most of the interactions between components take place between the components being located in the same place.

For example, a production chain has a number of places where the machines (static components) are located. Products (mobile components), processed by these machines, move (or are moved) from one place to another.

We can find in the literature several formalisms more or less dedicated to the modeling of such systems: the π-calculus [72], PEPA (Performance Evaluation Process Algebra) Nets [41], Colored Petri nets [53], etc.

Amongst AltaRica models, created so far, there are two main classes of models:

- Models of static systems. These simple but very huge models are basically assessed by automatic generation of Fault Trees.
- Models of dynamic systems. They are processed by automatic generation of sequences to find the failure scenarios and by stochastic simulation to calculate the reliability indicators.

However, the expressive power of AltaRica modeling language is not limited to these two classes of models. We therefore try to determine whether it is suitable for the modeling of systems with mobile components and whether the algorithms developed to assess AltaRica models could be used to evaluate the performance of such systems.

In the following article, we compare AltaRica and PEPA Nets [41] - a modeling formalism dedicated to systems with mobile components. A case study of a production system is used to illustrate the concepts of both formalisms. We also propose a modeling pattern to represent systems with mobile components using AltaRica 3.0 and present some experimental results.

Modeling Systems with Mobile Components: A comparison between AltaRica and PEPA nets

Leïla Kloul*[1,2], Tatiana Prosvirnova†[2] and Antoine Rauzy‡[2]

[1]PRiSM, Université de Versailles, 45 Avenue des États-Unis, 78000 Versailles, France
[2]LIX, Ecole Polytechnique, route de Saclay, 91128 Palaiseau, France

Abstract

Assessing the reliability of systems with mobile components, that is components whose locations and interactions change during the mission of the system, raises a number of specific modeling issues. In this article, we compare two candidate modeling formalisms to do so: AltaRica and PEPA nets. We study their respective advantages and drawbacks and we show benefits of a cross fertilization.

Keywords
Model-based safety analysis, modeling formalisms, mobility modeling, PEPA Nets, AltaRica.

1 Introduction

Many industrial systems embed mobile components, that is components whose locations and interactions change during the mission of the system. Systems of systems, like battlefields or mobile phone networks, enter obviously into this category. But mobile components can also be found in simpler systems, such as production chains. Assessing the reliability of systems with mobile components raises a number of specific modeling issues.

As an illustration, we consider in this article a simple plant that produces different types of goods within the same production chain. To calculate the reliability and other performance indicators on this system, one must be able to follow products individually, i.e. to capture dynamic behaviors such as dynamic change of locations of products and dynamic change of interactions between products and processing units.

We compare two candidate modeling formalisms to do so: PEPA nets [12] and AltaRica [7, 20]. These two formalisms have been developed by two different communities. Their "look-and-feel" are thus quite different. Yet, their underlying mathematical foundations are very similar: both rely on state automata and can be used to generate continuous-time Markov chains. It is therefore of interest to study their ability to assess the reliability of systems with mobile components, their respective advantages and drawbacks and to seek for opportunities of a cross fertilization.

PEPA nets combine the stochastic process algebra PEPA (Performance Evaluation Process Algebra [15]) with (stochastic) colored Petri nets [16]. Components are described as processes just as in PEPA, but they can additionally migrate from one place of the net to another, just as tokens in a colored Petri net. Components can interact (through synchronization of their actions) only if they are located in the same place. Operators behind PEPA nets remain simple to implement for dynamic creation/deletion of processes is not allowed. Any PEPA net model can be eventually compiled into a continuous-time Markov chain. The elegance of PEPA nets comes from its mathematical purity: only a very limited number of operators is sufficient to describe complex behaviors.

AltaRica has been designed with an engineering perspective. In AltaRica, the behavior of components is described by means of Guarded Transition Systems [18, 20]. Guarded Transition Systems (GTS)

*Leila.Kloul@prism.uvsq.fr
†Tatiana.Prosvirnova@polytechnique.edu
‡Antoine.Rauzy@lix.polytechnique.fr

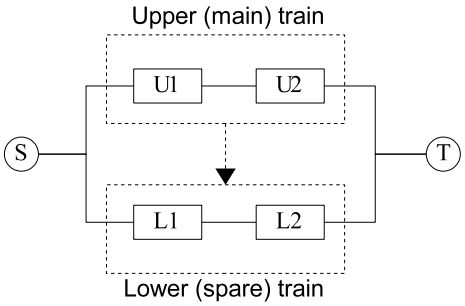


Figure 1: Production system.

generalize widely used formalisms such as Reliability Block Diagrams (see e.g. [4]) and Stochastic Petri nets [3]. Components can be assembled into hierarchies, their inputs and outputs can be connected and their transitions can be synchronized. Any hierarchical description can be "flattened" into a unique GTS. The semantics of a GTS is a Kripke structure (a reachability graph) that can be interpreted as a continuous-time Markov chain, under the condition that delays associated with transitions are exponentially distributed.

The richness of AltaRica makes it possible to design and to maintain industrial scale models [2, 6]. However, the previous versions of AltaRica embed no construct to model mobility. Since PEPA nets and AltaRica rely on similar mathematical foundations, it was worth to establish their respective strengths and weaknesses. This study resulted in the incorporation in the new version of AltaRica (AltaRica 3.0, still under specification) of the concept of guarded synchronization. This new concept unifies and simplifies previous AltaRica description of transitions and synchronizations and thus it eases modeling mobile components.

The contribution of this article is multiple. First, we examine, based on a simple example, the issues raised by the modeling of systems with mobile components. Second, we compare PEPA nets and AltaRica. We discuss their respective advantages and drawbacks. Third, we present the extension of AltaRica with the concept of guarded synchronization.

The remainder of this article is organized as follows. Section 2 presents the production system we shall use as a redwire throughout the article. Section 3 is dedicated to the related works. Section 4 and Section 5 present respectively PEPA nets and AltaRica and illustrate their philosophies by modeling (parts of) the production system. Section 6 compares the two approaches. Section 7 presents some experiences, performed to calculate reliability and performance indicators. Finally, Section 8 concludes the article.

2 Motivating Example

The example described in this section will be used as an illustration in the following sections.

2.1 Production System

We consider a production system made of two chains, as illustrated in Figure 1. The system is supplied by a source unit S. The upper chain, consisting of processing units U1 and U2 in series, is the main chain. The lower chain, consisting of processing units L1 and L2 in series, is a spare chain. The spare chain is normally used for other purposes but products can be rerouted to that chain in case the main chain is not available. The whole system supplies itself a target unit T. Units U1 and L1 on the one hand; U2 and L2 on the other hand play symmetrical roles. When either U1 or U2 fails, the other unit in the same chain is stopped and the lower chain is attempted to start. When both U1 and U2 are restarted (after their repair), the lower chain is stopped (or at least goes back to its primary purpose).

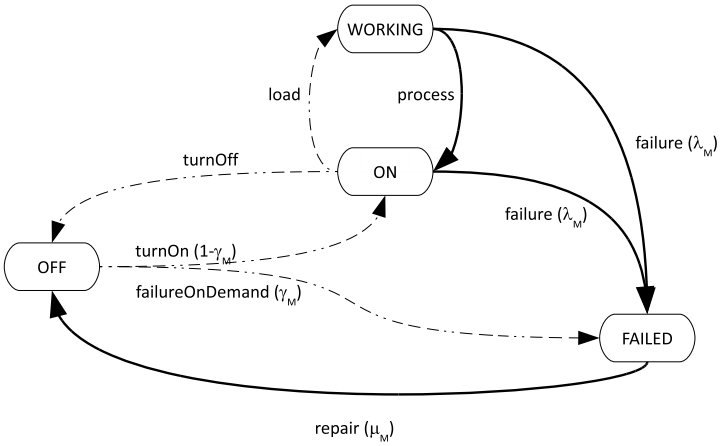


Figure 2: The Finite State Automaton modeling a Machine.

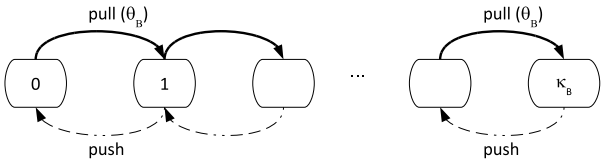


Figure 3: The Finite State Automaton modeling the board (of slots).

2.2 Production Units

For the sake of simplicity, source and target units are assumed to be perfect (and are never stopped). Each processing unit (U1, U2, L1 and L2) is composed of a machine M and a board B with a number κ_B of slots in which products are inserted. Source and target units can be seen simply as boards with slots.

All machines work (and fail) the same way. They can process only one product at a time. A machine M has a per hour failure rate λ_M (e.g. $1.0\ 10^{-3}$) when it is working. It is assumed not to fail when it is in a standby mode. When it is attempted to be turned on, it has a probability to fail on demand γ_M (e.g. 0.02), and thefore a probability $1 - \gamma_M$ to start correctly. It has a per hour repair rate μ_M (e.g. $2.0\ 10^{-1}$). It is assumed to be as good as new after a repair. When a machine fails the product it was processing (if any) needs to be reprocessed from scratch. Machines are not turned off when they are processing a product. The finite state automata modeling a machine is pictured in Figure 2. The machine starts to process a product, i.e. loads it, as soon as there is a non already processed product in a slot. The time taken by the processing of a product depends on the (type of the) product. On this figure, timed transitions are pictured with thick plain lines while instantaneous transitions are pictured with thin dashed lines. We shall keep this convention for subsequent figures.

The finite state automata modeling the board is pictured in Figure 3. For the sake of the simplicity, slots are not distinguished. Two actions can be performed on slots: pulling a product (from the previous unit) and pushing a product (to the next unit). Pulling and pushing a product are actually the two faces of the same action: transferring a product from one production unit to the next one. This action takes some time. We can assume without a loss of generality that the average transfer time t_B depends on the board B that "pulls" the product. A board B has therefore a per hour pulling rate $\theta_B = 1/t_B$ (e.g. $\theta_B = 100$).

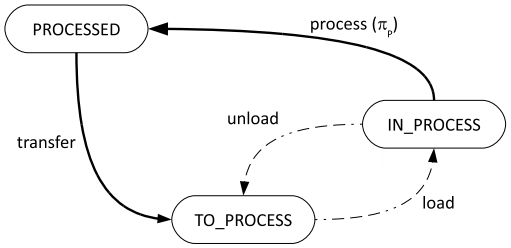


Figure 4: The Finite State Automaton modeling a product.

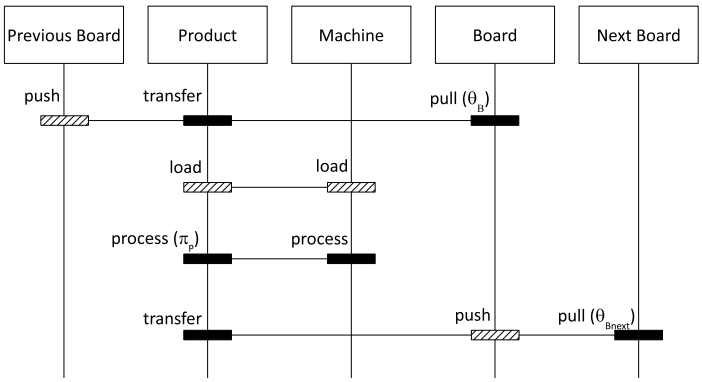


Figure 5: The Sequence Diagram for a successful processing of a product.

2.3 Products

Products are transferred from one unit to the next one. Once in the unit, a product can be either waiting to be processed, in process, or waiting to be transferred to the next unit. The average processing time p_P depends on the product so we have a processing rate $\pi_P = 1/p_P$ (e.g. $\pi_P = 10$). Figure 4 pictures the finite state automaton modeling a product. This automaton does not show the location of the product (which changes with the transition `transfer`).

2.4 Synchronizations/Simultaneity

We described so far individual behaviors of each component of the system. To complete the description, we need to describe which transitions are synchronized.

The sequence diagram pictured in Figure 5 shows synchronizations (horizontal lines) occurring in a successful processing sequence of a product. On this diagram, timed transitions are represented with plain rectangles, instantaneous transitions are represented with hatched rectangles.

The sequence diagram pictured in Figure 6 shows synchronizations occurring in a sequence in which a failure occurs during the processing of a product. Note that timed transitions may be synchronized with instantaneous transitions, e.g. the transition `failure` of the machine and the transition `unload` of the product. In this case, the resulting transition is indeed timed. The instantaneous transition takes place at the end of the timed action.

2.5 Wrap-Up

We want eventually to study the expected production of the system, throughout a given period of time and possibly additional performance indicators such as the mean down time of the main chain.

All the above hypotheses may be not very realistic. We just tried to concentrate into a small example a number of modeling issues:

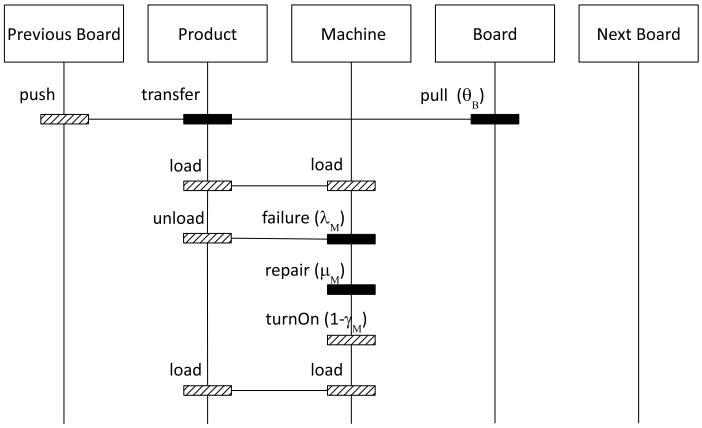


Figure 6: The Sequence Diagram for a sequence with a failure during the processing of a product.

- Products are mobile components. Some parameters, such as the processing rate, depend on products. Moreover, it may be worth to observe the individual trajectories of products.
- Products have to interact with processing units. These interactions can take place only in some locations and circumstances.
- The state of a processing unit depends on the states of other processing units, due to the command strategy of the system.
- Some transitions are instantaneous, some other take time. Timed transitions have rates that differ by orders of magnitude (the model is stiff).

The main modeling issue is to synchronize correctly actions of machines, boards and products and to do that for sufficiently many products while keeping the model tractable.

3 Related Works

Classical safety formalisms, such as Reliability Block Diagrams (RBD), Markov chains and Generalized Stochastic Petri Nets (GSPN) [3] are the most well known and widely used formalisms to assess reliability indicators of systems.

Boolean formalisms, such as RBD, are event based, naturally hierarchical and make it possible to describe remote interactions between components, i.e. flows of matter or information circulating through the system. They can be easily transformed into Fault Trees and assessed with very efficient algorithms (see e.g [19, 21]). However, Boolean formalisms put very strong constraints on events to be considered: they are assumed to be statistically independent, thus, it is not possible to take into account the order in which events occur any time.

States/transitions formalisms, such as Markov chains and Petri Nets, make it possible to capture dependencies amongst components, such as cold redundancies, resource sharing and sequences of actions. But it is quite difficult to use them, on the one hand, to represent remote interactions between components and on the other to compose seamlessly components into hierarchies.

Currently, rare are the modeling techniques that provide modeling mechanisms for systems with dynamic behaviors. Milner's π-calculus [17] is a paradigmatic formalism designed to capture dynamic behaviors, and probably one of the most widely studied. It has been developed to model communicating and mobile agents. It is very simple yet very powerful. However, its ability to create and to delete objects comes with a significant price in terms of assessment algorithms. This cost is so high that it is reasonably arguable whether such powerful formalism can be used for performance analyses. Like the π-calculus, PEPA-nets [8, 11–13] are a simple modeling technique that makes possible the description of migrations of components and of changes in their interactions. However, unlike π-calculus, the operators

Table 1: Comparison of Safety formalisms.

	RBD	Markov chains	GSPN	π-calculus	PEPA Nets	AltaRica DF*
Event based	▲	◯	◯	◯	◯	◯
Composition	⊙	■	⊙	⊙	⊙	◯
Hierarchical	◯	■	■	■	■	◯
Remote interactions	◯	■	■	▲	▲	◯
Mobility modeling	■	■	⊙	◯	◯	▲
Algorithm efficiency	◯	⊙	⊙	■	⊙	⊙

■ not suitable ▲ acceptable ⊙ good ◯ very good

* DF = Data-Flow

and the paradigms behind PEPA nets remain simple to implement for dynamic creation/deletion of objects is not allowed. As all state based modeling techniques, PEPA nets formalism remains prone to the problem of state space explosion. Moreover it does not provide a modeling mechanism to capture remote interactions amongst components.

Petri nets based techniques may be considered candidates to dynamic systems modeling. However, SPN models are constructed without explicit compositional structure regardless of the structure of the system being modeled. Consequently, subsequent techniques, such as Donatelli's Superposed GSPN [10] and Sanders' Stochastic Activity Networks [22] have aimed to provide mechanisms to represent the increasing complexity of the synchronization constraints of modern systems whilst retaining the compositional structure explicitly within the model. However, Petri nets based techniques do not provide an appropriate mechanism to capture dynamic change of interactions between objects as they do not allow the distinction between different contexts.

An extension of (non-stochastic) Petri nets which provides modeling concepts similar to PEPA nets is Valk's Elementary Object Systems (EOS) [23]. The tokens in an elementary object system are themselves Petri nets having individual dynamic behavior. However, like all Petri nets based formalism, EOS formalism suffers from a lack of an explicit compositional structure.

AltaRica DataFlow modeling language generalizes classical safety formalisms: it is a states/transitions formalism that allows hierarchical structuring of models and representing remote interactions. Nevertheless it would be quite difficult, from the practical modeling perspective, to use states/transitions formalisms to describe systems with mobile components, as presented in the example Section 2.

Table 1 summarizes this section.

4 Overview of PEPA nets

PEPA nets [12] combine the stochastic process algebra PEPA (Performance Evaluation Process Algebra) with stochastic colored Petri nets. This hybrid formalism is motivated by the observation that, in many systems, two distinct types of change of state can be identified: the *global* and *local* changes of states. The resulting formalism can be used to model applications such as mobile code systems where the PEPA terms are used to model the program code moving between the network hosts (the places in the net).

In the following, we first give an overview of the modeling language PEPA, then present the hybrid formalism PEPA nets.

4.1 PEPA

In PEPA [15], a system is described as an interaction of *components* which engage, either singly or multiply, in *activities*. These activities represent changes of state within a system. Each activity has an *action type* and a *duration* which is represented by the parameter of the associated exponential distribution: the *activity rate*. This parameter may be any positive real number, or the distinguished

symbol $\top$ (read as *unspecified*). Thus each activity is a pair (α, r) where α is the action type and r is the activity rate. We assume a countable set of components, denoted $\mathcal{C}$, and a countable set, $\mathcal{Y}$, of all possible action types. We denote by $\mathcal{A}ct \subseteq \mathcal{Y} \times \mathbb{R}^+$, the set of activities, where $\mathbb{R}^+$ is the set of positive real numbers together with the symbol $\top$.

PEPA provides a small but expressive set of combinators which allow expressions to be constructed defining the behavior of components, via the activities they undertake and the interactions between them.

Prefix $(\alpha, r).P$: this is the basic mechanism for constructing component behaviors. The component carries out activity (α, r) and subsequently behaves as component P.

Choice $P + Q$: the component may behave either as P or as Q: all the current activities of both components are enabled. The first activity to complete, determined by the race condition, distinguishes one component, the other is discarded.

Cooperation $P \underset{L}{\bowtie} Q$: the components proceed independently with any activities whose types do not occur in the *cooperation set* L (*individual activities*). However, activities with action types in the set L require the simultaneous involvement of both components (*shared activities*). When the set L is empty, we use the more concise notation $P \parallel Q$ to represent $P \underset{\emptyset}{\bowtie} Q$.

The published stochastic process algebras differ on how the rate of shared activities are defined [14]. In PEPA the shared activity occurs at the rate of the slowest participant. If an activity has an unspecified rate, denoted $\top$, the component is *passive* with respect to that action type. This means that the component does not influence the rate at which any shared activity occurs.

Hiding P/L: the component behaves as P except that any activities of types within the set L are *hidden*, i.e. they exhibit the unknown type τ and can be regarded as an internal delay by the component. These activities cannot be carried out in cooperation with another component.

Constant $A \stackrel{def}{=} P$: Constants are components whose meaning is given by a defining equation. $A \stackrel{def}{=} P$ gives the constant A the behavior of component P. This is how we assign names to components (behaviors).

The evolution of a PEPA model is governed by the Structured Operational Semantics (SOS) rules of the language. These rules define the admissible transitions or state changes associated with each combinator. They give rise to a multi-labeled transition system or derivation graph from which a continuous-time Markov chain can be derived.

Example: Consider the upper train of the production system described in Section 2. If we want to model the first processing unit, U_1, one can use two PEPA components, namely $Machine_1$ and $Product$. The first component models the behavior of the machine in the processing unit, whereas the second one models the items provided by the source, in order to be processed by U_1.

- Component $Machine_1$: when the processing unit is working properly, it first loads a new item if this one is available. This is modeled using action type $load_1$ which rate l_1 is supposed to be high as the action of loading is assumed to be instantaneous. Once the item loaded, three different events may occur: the processing of the item, a failure of the machine, or the arrival of an order for the machine to be switched off because U_2, the other processing unit in the main train, is in the failure state. The three events are modeled using action types $process_1$, $failure_1$ and $turnOff_1$, respectively. In the failure state $Machine_{1\_FAILED}$, the machine can be either repaired (action type $repair_1$), or receive a $turn\ off$ order that will not make the component change state. When repaired, the machine is stopped (state $Machine_{1\_OFF}$) until a switch on order is received. This is modeled using activity $(turnOn_1, 1 - \gamma_{u_1})$, whereas the occurrence of a failure on demand is modeled using activity $(failureOnDemand_1, \gamma_{u_1})$. The initial state of the component is $Machine_{1\_ON}$. Thus,

the complete component is defined as follows:

$$\begin{aligned}
Machine_{1_ON} &\stackrel{def}{=} (load, l_1).Machine_{1_WORKING} \\
&+ (failure_1, \lambda_{u_1}).Machine_{1_FAILED} \\
&+ (turnOff_1, s_1).Machine_{1_OFF} \\
Machine_{1_WORKING} &\stackrel{def}{=} (process_1, \pi_p).Machine_{1_ON} \\
&+ (failure_1, \lambda_{u_1}).Machine_{1_FAILED} \\
Machine_{1_FAILED} &\stackrel{def}{=} (repair_1, \mu_{u_1}).Machine_{1_OFF} \\
&+ (turnOff_1, s_2).Machine_{1_FAILED} \\
Machine_{1_OFF} &\stackrel{def}{=} (turnOn_1, s_2 \times (1 - \gamma_{u_1})).Machine_{1_ON} \\
&+ (failureOnDemand_1, s_0 \times \gamma_{u_1}).Machine_{1_FAILED}
\end{aligned}$$

As loading a product to be processed is assumed to be an instantaneous action and does not really make the machine change state, we do not consider it in the PEPA model. However, we have to take into account the instantaneous events *failure on demand*, *turn off* and *turn on*, because they make the machine change states. For that we suppose that these actions occur at the rates s_0, s_1 and s_2, respectively.

- Component *Product*: it can be defined by the loading and processing undertaken in unit U_1 as follows:

$$\begin{aligned}
Product &\stackrel{def}{=} (load_1, \top).Product' \\
Product' &\stackrel{def}{=} (process_1, \top).Product
\end{aligned}$$

  In this component the rates associated with action types $load_1$ and $process_1$ are unspecified; component $Machine_1$ specifies the rate at which the loading and the processing occur.

- The behavior of the complete processing unit U_1 is modeled using the PEPA equation which specifies that components $Machine_1$ and *Product* must cooperate (synchronize) on action types $load_1$ and $process_1$.

$$U_1 \stackrel{def}{=} Machine_{1\_ON} \underset{\{load_1, process_1\}}{\bowtie} Product$$

4.2 PEPA nets

As PEPA nets combine PEPA with colored stochastic Petri nets, two types of change of state are possible: the *transitions* of PEPA components and the *firings* of the net. Transitions of PEPA components will typically be used to model small-scale (local) changes of state as components undertake activities. Firings of the net will be used to model macro-step (global) changes of state such as context switches or mobile software agents moving from one network host to another.

A PEPA net is made up of PEPA *contexts*, one at each place in the net. A context consists of a number of *static* components (possibly zero) and a number of *cells* (at least one). A cell is a storage area dedicated to storing a PEPA component of the specified type. The components which fill cells are the *mobile* components and can circulate as the *tokens* of the net. In contrast, the static components cannot move.

The mobile components or tokens of a PEPA net are terms of the PEPA stochastic process algebra which define the behavior of components via the activities they undertake and the interactions between them. Thus each token has a type given by its definition. This type determines the transitions and firings which a token can engage in. It also restricts the places in which it may be, since it may only enter a cell of the corresponding type.

As the firings, on the one hand, and the transitions, on the other hand, are special cases of PEPA activities, we differentiate the action types associated with each of these. We denote by $\mathcal{Y}_f$ the set of action types at the net level and by $\mathcal{Y}_t$ the set of action types inside the places such that $\mathcal{Y} = \mathcal{Y}_f \cup \mathcal{Y}_t$. Similarly, we denote by $\mathcal{A}ct_t \subseteq \mathcal{Y}_t \times \mathbb{R}^+$ the set of activities undertaken by the components inside the places and by $\mathcal{A}ct_f \subseteq \mathcal{Y}_f \times \mathbb{R}^+$ the set of activities at the net level such that $\mathcal{A}ct = \mathcal{A}ct_f \cup \mathcal{A}ct_t$.

Definition 4.1 *A PEPA net $\mathcal{V}$ is a tuple $\mathcal{V} = (\mathcal{P}, \mathcal{T}, I, O, \ell, \pi, \mathcal{F}_P, K, M_0)$ such that*

- *$\mathcal{P}$ is a finite set of places;*
- *$\mathcal{T}$ is a finite set of net transitions;*
- *$I : \mathcal{T} \to \mathcal{P}$ is the input function;*
- *$O : \mathcal{T} \to \mathcal{P}$ is the output function;*
- *$\ell : \mathcal{T} \to (\mathcal{Y}_f, \mathbb{R}^+ \cup \{\top\})$ is the labeling function, which assigns a PEPA activity ((type, rate) pair) to each transition. The rate determines the negative exponential distribution governing the delay associated with the transition;*
- *$\pi : \mathcal{Y}_f \to \mathbb{N}$ is the priority function which assigns priorities (represented by natural numbers) to firing action types;*
- *$\mathcal{F}_P : \mathcal{P} \to P$ is the place definition function which assigns a PEPA context, containing at least one cell, to each place;*
- *K is the set of token component definitions;*
- *M_0 is the initial marking of the net.*

PEPA net behavior is governed by structured operational semantic rules. These consist of the original rules for PEPA and some additional rules capturing the meaning of a cell, as well as the enabling and firing rules of the net level structure [12]. The states of the model are the marking vectors, which have one entry for each place of the PEPA net. The semantic rules govern the possible evolution of a state, giving rise to a labeled transition system or derivation graph. The nodes of the graph are the marking vectors and the activities (individual, shared or firing activities) give the arcs of the graph. This graph gives rise to a CTMC which can be solved to obtain a steady-state probability distribution from which performance measures can be derived.

The syntax of PEPA nets is given in Figure 7.

$$N ::= D^+ M \qquad \text{(net)}$$

(definitions and marking)

$$M ::= (M_{\mathbf{P}}, \ldots) \qquad \text{(marking)}$$

$$M_{\mathbf{P}} ::= \mathbf{P}[C, \ldots] \qquad \text{(place marking)}$$

(marking vectors)

$$D ::= I \stackrel{def}{=} S \qquad \text{(component defn)}$$

$$\mid \quad \mathbf{P}[C] \stackrel{def}{=} P[C] \qquad \text{(place defn)}$$

$$\mid \quad \mathbf{P}[C, \ldots] \stackrel{def}{=} P[C] \underset{L}{\bowtie} P \qquad \text{(place defn)}$$

(identifier declarations)

$$S ::= (\alpha, r).S \qquad \text{(prefix)}$$

$$\mid \quad S + S \qquad \text{(choice)}$$

$$\mid \quad I \qquad \text{(identifier)}$$

(sequential components)

$$P ::= P \underset{L}{\bowtie} P \qquad \text{(cooperation)}$$

$$\mid \quad P/L \qquad \text{(hiding)}$$

$$\mid \quad P[C] \qquad \text{(cell)}$$

$$\mid \quad I \qquad \text{(identifier)}$$

(concurrent components)

$$C ::= \text{`}\_\text{'} \qquad \text{(empty)}$$

$$\mid \quad S \qquad \text{(full)}$$

(cell term expressions)

Figure 7: The syntax of PEPA nets.

In the grammar S denotes a *sequential component* and P denotes a *concurrent component* which executes in parallel. I stands for a constant which denotes either a sequential or a concurrent component, as bound by definition.

Example: Consider the sub-system composed of source S, processing units U_1 and U_2, and the target unit T of the production system described in Section 2. The PEPA net model of this sub-system consists of four places: $SOURCE$, $MAIN\_UNIT_1$, $MAIN\_UNIT_2$ and $TARGET$. The net structure of the model is depicted in Figure 8.

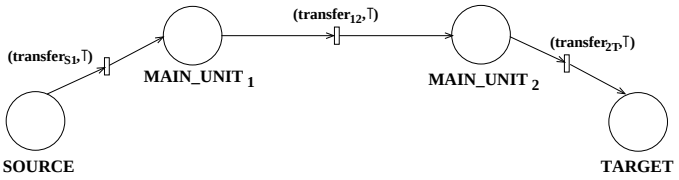


Figure 8: The PEPA net model.

- Place $SOURCE$: it models the source which supplies unit U_1 with the items to process. Thus we model each of these items using component $Product$ (see Section 4.1) which now is a mobile component since each item has to go through all the units of the system to be completed. Thus the definition of component $Product$ is enriched with activities that model the movements of the component between the places. These activities label the firing transitions on the net structure (Figure 8).

$$\begin{aligned}
Product &\stackrel{def}{=} (\mathbf{transfer_{S1}}, \top).Product_1 \\
Product_1 &\stackrel{def}{=} (load_1, \top).Product'_1 \\
Product'_1 &\stackrel{def}{=} (process_1, \top).Product_2 \\
Product_2 &\stackrel{def}{=} (\mathbf{transfer_{12}}, \top).Product_3 \\
Product_3 &\stackrel{def}{=} (load_2, \top).Product'_3 \\
Product'_3 &\stackrel{def}{=} (process_2, \top).Product_4 \\
Product_4 &\stackrel{def}{=} (\mathbf{transfer_{2T}}, \top).Product
\end{aligned}$$

  Place $SOURCE$ consists solely of the items to provide to unit U_1. Thus initially all the items are in this place.

$$SOURCE[\_, \ldots, \_] \stackrel{def}{=} Product[Product] || \ldots || Product[Product]$$

- Place $MAIN\_UNIT_1$: it models the processing unit U_1. The behavior of the corresponding processing machine is modeled using static component $Machine_1$ (see Section 4.1).

  The whole place is then modeled as the interaction of $Machine_1$ and mobile component $Product$ on action type $process_1$. The maximum number of mobile components in the place corresponds to the storage capacity of unit U_1, that is κ_{U_1}.

$$MAIN\_UNIT_1[\_, \ldots, \_] \stackrel{def}{=} Machine_{1\_ON} \underset{\{process_1\}}{\bowtie} (Product[\_] || \ldots || Product[\_])$$

- Place $MAIN\_UNIT_2$: it models the behavior of processing unit U_2 which has the same behavior as U_1. Thus we use a similar component, namely $Machine_2$, to model the processing machine in

U_2.

$$
\begin{array}{lcl}
Machine_{2\_ON} & \stackrel{def}{=} & (load_2, l_2).Machine_{2\_WORKING} \\
& + & (failure_2, \lambda_{u_2}).Machine_{2\_FAILED} \\
& + & (turnOff_2, s'_1).Machine_{2\_OFF} \\
Machine_{2\_WORKING} & \stackrel{def}{=} & (process_2, \pi_p).Machine_{2\_ON} \\
& + & (failure_2, \lambda_{u_2}).Machine_{2\_FAILED} \\
Machine_{2\_FAILED} & \stackrel{def}{=} & (repair_2, \mu_{u_2}).Machine_{2\_OFF} \\
& + & (turnOff_2, s'_2).Machine_{2\_FAILED} \\
Machine_{2\_OFF} & \stackrel{def}{=} & (turnOn_2, s'_2 \times (1 - \gamma_{u_2})).Machine_{2\_ON} \\
& + & (failureOnDemand_2, s'_0 \times \gamma_{u_2}).Machine_{2\_FAILED}
\end{array}
$$

The complete place is then modeled as the cooperation of $Machine_{2\_ON}$ and $Product$ on action types $load_2$ and $process_2$.

$$MAIN\_UNIT_2[\_, \ldots, \_] \stackrel{def}{=} Machine_{2\_ON} \underset{\{load_2, process_2\}}{\bowtie} (Product[\_] || \ldots || Product[\_])$$

Similarly to $MAIN\_UNIT_1$, the maximum number of components $Product$ in $MAIN\_UNIT_2$ is κ_{U_2}, the storage capacity of unit U_2.

- Place $TARGET$: it models the target unit and consists solely of the finished items arriving from unit U_2. Initially, it is empty.

$$TARGET[\_, \ldots, \_] \stackrel{def}{=} Product[\_] || \ldots || Product[\_]$$

Note that the maximum number of components $Product$ in places $SOURCE$ and $TARGET$ is defined by storage capacity κ_S and κ_T, respectively.

5 AltaRica Overview

AltaRica is a high level modeling language initially dedicated to safety analysis. The first version of AltaRica modeling language has been developed in LaBRI in ninetieth [5]. A few years later, a second (data-flow) version has been developed to handle industrial scale models that the first version, too expressive, was inefficient to tackle. A number of processing tools have been developed for AltaRica such as compilers to Fault Trees, compilers to Markov chains, generators of critical sequences, model-checkers and stochastic simulators. Several Integrated Modeling Environments use AltaRica as their internal representation language.

The third version (AltaRica 3.0) is under specification at the time we write these lines. AltaRica 3.0 will be a major evolution of the language (and the processing tools). This new version integrates notions of object-oriented programming languages such as inheritance and prototypes. It improves the reusability of components and knowledge capitalization. It adds also the ability to handle looped systems. The models presented below are written in AltaRica 3.0. The formal semantics of AltaRica 3.0 is based on the notion of Guarded Transition Systems - a states/events formalism defined in Reference [20].

5.1 Guarded Transition Systems

Guarded Transition Systems, GTS for short, are input/output automata. The state space is described implicitly as, for instance, in Petri nets. We shall introduce here GTS by means of an example. Consider first the automaton for the machine pictured Figure 2. The AltaRica code for this automaton is given Figure 9.

```
domain MachineState { OFF, ON, WORKING, FAILED }

class Machine
  MachineState state (init = OFF);
  Boolean demanded (reset = false);
  event turnOn (delay = 0, expectation = 1 - gamma);
  event failureOnDemand (delay = 0, expectation = gamma);
  event failure (delay = exponential(lambda));
  event repair (delay = exponential(mu));
  event turnOff (delay = 0);
  event load (delay = 0);
  event process;
  parameter Real gamma = 0.02;
  parameter Real lambda = 0.001;
  parameter Real mu = 0.1;
transition
  turnOn:  state==OFF and demanded -> state := ON;
  failureOnDemand:  state==OFF and demanded -> state := FAILED;
  turnOff:  state==ON and not demanded -> state := OFF;
  failure:  state==ON or state==WORKING -> state := FAILED;
  repair:  state==FAILED -> state := OFF;
  load:  state==ON -> state := WORKING;
  process:  state==WORKING -> state := ON;
end
```

Figure 9: The AltaRica code for the Finite State Automaton modeling a machine.

Variables: The internal state of the machine is represented by means of the state variable `state`. `state` takes its value in the domain `MachineState` declared upfront. The initial values of state variables (there may be several) are specified by means of the attribute `init`.

Another variable is declared: `demanded`. This variable is a Boolean flow variable. It is used to implement the command, i.e. to tell when to turn on and off the machine. Values of flow variables are reset after each transition firing, then updated by means of an assertion. This mechanism will be described latter. From a syntaxic viewpoint, flow variables are introduced (and distinguished from state variables) by means of the attribute `reset`.

Events: The state of the machine changes under the occurrence of an event. Events are introduced with the keyword `event`. A delay is associated with each event by means of the attribute `delay`. In our example, delays of events `failure` and `repair` are random variables exponentially distributed with respective rates `lambda` and `mu`. In other words, they obey a Markovian hypothesis. Events `turnOn` and `failureOnDemand` are instantaneous (their delay is 0). Both are fireable when the machine is `OFF`. `turnOn` has the probability `1 - gamma` to be fired while `failureOnDemand` has a probability `gamma` to be fired in this state. This probability is given through the attribute `expectation`. Delays of events `load` and `process` are left unspecified.

Transitions: A transition is a triple $\langle e, G, P \rangle$, also denoted $e : G \rightarrow P$, where e is an event, G is a Boolean expression, so-called the guard (or the pre-condition) of the transition, P is an instruction, so-called the action (or the post-condition) of the transition. Transitions are described in the clause `transitions`. In the example above if the state of the processing machine is `WORKING`, then two transitions are fireable: the transition labeled with the event `failure` and the transition labeled with the event `process`. If the delay drawn for the transition `failure` is the shortest, then this transition is fired and its action is executed: `state` is switched to `FAILED`.

Parameters: Parameters are constant values that come with the definition of the GTS. When the GTS is instantiated, their values may be changed. In the above example, there are three parameters `gamma`, `lambda` and `mu` that define respectively the probability of failure on demand and the failure and repair rates.

5.2 Composition and Synchronization

The AltaRica code for the automaton describing the board (as pictured Figure 3) is given Figure 10. This code deserves no additional explanation.

```
class Board
  Integer count (init = 0);
  event pull (delay = exponential(theta));
  event push (delay = 0);
  parameter Integer capacity = 3;
  parameter Real theta = 60;
transition
  pull:  count<capacity -> count := count+1;
  push:  count>0 -> count := count-1;
end
```

Figure 10: The AltaRica code for the Finite State Automaton modeling the board.

Now we can consider the model for a processing unit. AltaRica 3.0 is an object oriented language. Therefore, the AltaRica class that describes a processing unit embeds an instance of the class describing the machine and an instance of the class describing the board, as illustrated Figure 11.

```
class ProcessingUnit
  Machine M;
  Board B;
transition
  M.load:  !M.load & B.count>0 -> skip;
end
```

Figure 11: The AltaRica code for a Processing Unit.

This combination is however not a mere product: the machine cannot load a product if the board is empty. This additional constraint is described by means of a synchronization. The transition `load` of the machine `M` (`M.load`) is synchronized with an anonymous transition that just checks that `B.count` is positive and does nothing (its action is the empty instruction `skip`). This synchronization creates a new transition. This transition is obtained by and-ing the guards of synchronized transitions and composing their actions. In our case, the synchronization creates the following transition.

```
M.state==ON and B.count>0 -> M.state := WORKING;
```

A synchronization can involve any number of transitions. Transitions involved in a synchronization cease to exist individually. It is the case here for the transition `M.load`. Since we don't want to create a fresh event for the created transition, we use the event `M.load`. Note finally that `M.load` is prefixed with an exclamation mark (`!`). This modality indicates that the individual transition `M.load` is mandatory for the synchronized transition to be fired. Anonymous transitions are always mandatory. The modality `?` makes it possible to synchronize transitions only when they are fireable. We won't describe it here fully for we shall not use it in our model.

5.3 Flow Variables and Assertions

In the AltaRica code for the machine, given Figure 9, the flow variable `demanded` is used to guard the instantaneous transitions `turnOn` and `turnOff`. Conversely to state variables, that are initialized at the beginning of a run and then modified through actions of transitions, the value of flow variables are recalculated after each transition firing. This recalculation is performed by means of assertions. Assertions are instructions just as actions of transitions. The difference stands in that actions of transitions assign state variables only while assertions assign flow variables only. Moreover, each component has a unique assertion that is applied after each transition firing.

Most of AltaRica models make a great use of flow variables and assertions. They are used to model information flows circulating through a system. They may represent physical connections between components, control commands, fluid circulation, electric power, etc. They offer an easy and elegant way to express dependencies on external factors.

In our example, we shall use them to a limited extent, in order to implement the command strategy. The AltaRica code for the plant is pictured Figure 12. This code composes four processing units. When one of the two main units fails, the other one must be stopped (possibly after finishing to process a product) and the production must be switched to the spare line. Conversely, both units of the main line are attempted to start as soon as they are `OFF`, i.e. after a repair.

```
class Plant
  Board S, T;
  ProcessingUnit U1, U2, L1, L2;
  Boolean mainLineFailed, spareLineFailed (reset = false);
assertion
  mainLineFailed := U1.M.state==FAILED or U2.M.state==FAILED;
  spareLineFailed := L1.M.state==FAILED or L2.M.state==FAILED;
  U1.M.demanded := not mainLineFailed;
  U2.M.demanded := U1.M.demanded;
  L1.M.demanded := mainLineFailed and not spareLineFailed;
  L2.M.demanded := L1.M.demanded;
end
```

Figure 12: The AltaRica code for the Plant.

5.4 Mobile Components

It remains now to model products and to synchronize them with the plant. The AltaRica code for products is given Figure 13. It implements the automaton pictured Figure 4.

Now, we have to synchronize the plant with a number of products. Figure 14 shows a part of the code to do so. This code uses the same mechanism of synchronization as previously, except that more transitions are involved in synchronizations.

6 Comparison of two approaches

Both PEPA nets and AltaRica rely heavily on state automata, but are quite different in the way they represent them. To some extent, whether to use a Process Algebra style or a Guarded Transition Systems style is a matter of taste. Guarded Transition Systems are probably more powerful and more compact, thanks to the use of state variables. Aside the way automata are encoded, there are two main differences between the two formalisms:

- AltaRica embeds the concept of flow variables. Flow variables (and assertion) make it possible to describe remote interactions between components. Modeling such interactions with PEPA nets is more complex, although possible.

```
domain ProductState { PROCESSED, TO_PROCESS, IN_PROCESS }
domain Location { SOURCE, UNIT_U1, UNIT_U2, UNIT_L1, UNIT_L2, TARGET }

class Product
  ProductState state (init = PROCESSED);
  Location location (init = SOURCE);
  event transfer;
  event load (delay = 0);
  event unload (delay = 0);
  event process (delay = exponential(pi));
  parameter Real pi = 10;
transition
  transfer:  state==PROCESSED -> state := TO_PROCESS;
  load:  state==TO_PROCESS -> state := IN_PROCESS;
  unload:  state==IN_PROCESS -> state := TO_PROCESS;
  process:  state==IN_PROCESS -> state := PROCESSED;
end
```

Figure 13: The AltaRica code for the Products.

```
class System
  Plant plant;
  Product p1; Product p2; Product p3; ...
transition
  ⋮
  plant.U1.B.pull:
    !plant.U1.B.pull & !p3.transfer & p3.location==SOURCE -> p3.location := UNIT_U1;
  plant.U1.M.load:
    !plant.U1.M.load & !p3.load & p3.location==UNIT_U1 -> skip;
  plant.U1.M.process:
    !plant.U1.M.process & !p3.process & p3.location==UNIT_U1 -> skip;
  plant.U1.M.failure:
    !plant.U1.M.failure & !p3.unload & p3.location==UNIT_U1 -> skip;
  ⋮
end
```

Figure 14: The AltaRica code for the System.

- PEPA nets provides a mechanism to describe component locations, while such a mechanism has to be modeled in AltaRica.

In the remainder of this section, we shall examine both issues.

6.1 Modeling Remote Interactions between Components in PEPA nets

In PEPA nets, there is no direct means to express behavioral dependencies between components located in different places for components can cooperate or synchronize only if they are in the same place. Thus the notion of flows does not exist as a such. However, it is always possible to model systems with flows. To do so we have to introduce in the model mobile components which are, a priori, unnecessary, but which allow us to express remote interactions between components. The following example is a good illustration of that.

Example: Consider the system used in the example of Section 4.2. In order to complete modeling the upper train of the example, actions $turnOff$ and $turnOn$ have to be synchronizing actions. Indeed both machine units $U1$ and $U2$ must be stopped if one of them fails and both must be restarted if the failure is repaired. Thus, we assume that the production system has a control center in charge of generating the stoppage/restart orders. When a failure occurs at the upper train, the control center has to send a stoppage signal to the working unit in the main train, and a start signal to the units in the spare train. Once the failed unit is repaired, the control center has to stop the spare train while starting the upper train again.

The net structure of the corresponding PEPA net model is depicted in Figure 15. This structure consists of the net structure in Figure 8 to which a new place, namely $CENTER$, has been added. This place models the control center of the production system.

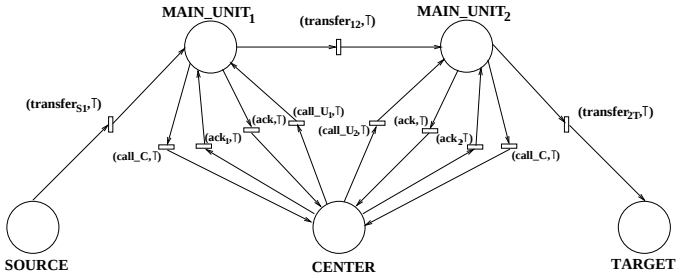


Figure 15: The net structure of the PEPA net model.

In the new PEPA net model, the definitions of places $SOURCE$ and $TARGET$ remain unchanged. However, the definitions of places $MAIN\_UNIT_1$ and $MAIN\_UNIT_2$ have to be changed in order to take into account their interactions with the control center. In the case of the interaction between $MAIN\_UNIT_1$ and $CENTER$, we use two mobile components, $Signal\_1$ and $Signal\_C$. The role of the former is to inform the control center about the state of the machine in $U1$ (failed, repaired). The reception of this information is then acknowledged to $U1$, using the same mobile component ($Signal\_1$). The latter is sent by the control center to $U1$ in order to stop/start its machine in the case of the failure/repair of the machine in $U2$. Thus $MAIN\_UNIT_1$ is defined as follows.

$$\begin{array}{lcl} MAIN\_UNIT_1[\_,\ldots,\_] & \stackrel{def}{=} & ((Signal\_C[\_]||Signal\_1[Signal\_1]) \underset{L}{\bowtie} Machine_{1\_ON}) \\ & \underset{\{process_1\}}{\bowtie} & (Product[\_]||\ldots||Product[\_]) \end{array}$$

where $L = \{turnOn_1, turnOff_1, failureOnDemand_1\}$ and mobile component $Signal\_1$ has the following behavior.

$$\begin{array}{lcl} Signal\_1 & \stackrel{def}{=} & (failure_1, \top).Signal\_1_1 \\ Signal\_1_1 & \stackrel{def}{=} & (\mathbf{call\_C}, e_1).Signal\_1_2 \\ Signal\_1_2 & \stackrel{def}{=} & (failed_1, e_2).Signal\_1_3 \\ Signal\_1_3 & \stackrel{def}{=} & (\mathbf{ack_1}, e_4).Signal\_1_4 \\ Signal\_1_4 & \stackrel{def}{=} & (repaire_1, \top).Signal\_1_5 \\ Signal\_1_5 & \stackrel{def}{=} & (\mathbf{call\_C}, e_1).Signal\_1_6 \\ Signal\_1_6 & \stackrel{def}{=} & (repaired_1, e_2).Signal\_1_7 \\ Signal\_1_7 & \stackrel{def}{=} & (\mathbf{ack_1}, e_4).Signal\_1 \end{array}$$

As specified in the equation of $MAIN\_UNIT_1$, component $Signal\_1$ is initially located in place $MAIN\_UNIT_1$. The behavior of component $Signal\_C$, which is initially located in place $CENTER$, is

Table 2: Comparison of Safety formalisms: AltaRica 3.0.

	PEPA Nets	AltaRica DataFlow	AltaRica 3.0
Event based	◯	◯	◯
Composition	⊙	◯	◯
Hierarchical	■	◯	◯
Remote interactions	▲	◯	◯
Mobility modeling	◯	▲	⊙
Algorithm efficiency	⊙	⊙	⊙

■ not suitable ▲ acceptable ⊙ good ◯ very good

defined as follows:

$$
\begin{aligned}
Signal\_C &\stackrel{def}{=} (failed_2, \top).Signal\_C_1 + (repaired_2, \top).Signal\_C_1 \\
Signal\_C_1 &\stackrel{def}{=} (\mathbf{call\_U_1}, e_1).Signal\_C_2 \\
Signal\_C_2 &\stackrel{def}{=} (turnOff, e_2).Signal\_C_3 + (turnOn, e_3).Signal\_C_3 \\
Signal\_C_3 &\stackrel{def}{=} (\mathbf{ack}, e_4).Signal\_C
\end{aligned}
$$

Similarly place $MAIN\_UNIT_2$ is changed in order to include two mobile components, namely $Signal\_2$ and $Signal'\_C$. These components are similar to $Signal\_1$ and $Signal\_C$, respectively. Thus, place $CENTER$ can be modeled as the interaction between the four mobiles components as follows:

$$CENTER[\_, \ldots, \_] \stackrel{def}{=} (Signal\_C[\ Signal\_C\ ] \mid\mid Signal'\_C[\ Signal'\_C\ ]) \underset{M}{\bowtie} (Signal\_1[\ \_\ ] \mid\mid Signal\_2[\ \_\ ])$$

where cooperation set $M = \{failed_1, repaired_1, failed_2, repaired_2\}$.

This example shows that it is always possible de model remote interactions between components located in different places. However, it comes at a certain price as it requires the use of additional components, which leads to the increase of the model size.

6.2 Modeling Mobility with AltaRica

AltaRica provides no specific construct to model mobility. The location of a component can be modeled as symbolic state variable. In the code presented Figure 13, the type of this variable is an user declared domain. It would be also possible to declare it just as `Symbol`, the set of all symbolic constants. In this way, the topography of the underlying network could be changed without changing the code for products.

PEPA nets synchronize events on their names, so that many components can be synchronized by means of a single rule. AltaRica requires to write down each synchronization explicitly, as sketched Figure 14. Writing all synchronizations for all products by hand would be both tedious and error prone, even if the concept of guarded synchronization, introduced in AltaRica thanks to the present study, simplifies greatly the task. We are presently using scripts (typically written in Python or Pearl) to generate automatically synchronizations. In the future, some specific constructs or some meta-modeling facilities should be added to the language in order to avoid to use external tools.

Table 2 summarizes this section.

7 Experiments

We have performed some experiments with AltaRica and PEPA Nets models of the motivating example.

A continuous time Markov chain (CTMC) has been generated from the AltaRica model. This generation is done in the following way. First, the AltaRica model is flattened into a unique Guarded Transition System [18, 20]. Second, the corresponding reachability graph is generated. Indeed, the semantics of a

Mobile products	AltaRica		PEPA Nets	
	MC states number	Running time	MC states number	Running time
1	155	0.01 sec.	982936	1159 sec.
2	2473	0.62 sec.	-	-
3	37379	257 sec.	-	-
4	-	-	-	-

Table 3: Experiments.

Mobile products	Time	Reliability	Availability
1	24h	0.996656	0.999616
2	24h	0.992918	0.999156
3	24h	0.988702	0.998616

Table 4: Availability and Reliability.

Guarded Transition System is a Kripke structure (a reachability graph) with nodes defined by variable assignments (i.e. variables and their values) and edges defined by transitions and labeled by events. If the delays associated with the events are exponentially distributed, than the reachability graph can be interpreted as a continuous time Markov chain. In case when the graph contains immediate transitions (delays associated with labeling events are equal to 0), they are just collapsed using the fact that an exponential delay with rate λ followed by an immediate transition of probability p is equivalent to a transition with an exponential delay of rate $p\lambda$.

Similarly, we have generated a continuous time Markov chain from the PEPA nets model, using the PEPA Workbench for PEPA nets models [1]. The semantic rules governing the possible evolution of a state, give rise to a multi-labelled transition system or derivation graph. The nodes of the graph are the marking vectors and the activities (individual, shared or firing activities) give the arcs of the graph. This graph gives rise to a CTMC.

As it is shown in Table 3, the size of the generated Markov chains grows exponentially with the number of mobile products, and this in both cases. However, the problem of exponential growth is more striking in the case of PEPA nets models. Indeed for a model containing only one mobile product, the generated Markov chain has almost one million states. For more than one product, the PEPA Workbench just could not generate the associated Markov chain. This is due to the fact that flows cannot be represented implicitly with PEPA nets; they have to be explicitly modeled using additional components. In our model, four components, with eight, five, eight and nine states respectively, have been added into the model, in order to take into account the flows in the system. Moreover, unlike in AltaRica model, we had to model explicitly the control center by adding a place in the model, and consequently increasing the model size.

The generation of Markov chains seems to be hardly usable in case of such a complex model. It might be promising to generate an approximated Markov chain as proposed in [9].

The generated Markov chains can be assessed with specific tools in order to calculate performance indicators. Some results of system availability and reliability, calculated for the AltaRica model are given in Table 4.

It would be more interesting and efficient in our case to perform stochastic simulations of models. Thus, the model captures not only failures and repairs of components, but also their functional behavior (e.g. pulling, processing and loading of products), we need to focus on short periods of time (e.g. 24h against 10000h in traditional reliability studies) to calculate performance indicators.

8 Conclusion

In this article, we showed that assessing the reliability of systems with mobile components raises a number of specific modeling issues. Most of these issues stand in the modeling of interactions between components: these interactions can take place only under certain conditions, but many different components can exhibit the same behavior.

We investigated the relationship between PEPA nets, a performance modeling process algebra, and AltaRica, an engineering oriented modeling language for safety analysis. These formalisms rely on very similar mathematical foundations: they are based on finite state automata and they can be compiled into Markov chains. Thus, we have sought to compare their expressiveness at the modeling, rather than at the Markovian, level. Our comparison revealed that AltaRica provides no direct mechanisms for mobility modeling, in particular it does not allow modeling location dependent synchronizations. Thus we have enhanced AltaRica by incorporating this modeling construct and showed that it offers increased flexibility to the modeler. On the other way round, while the flow in a system can be naturally modeled with AltaRica, PEPA nets provide no direct modeling mechanisms for it. The net structure prevents a direct modeling of remote interactions between components located in different places.

References

[1] http://www.dcs.ed.ac.uk/pepa/tools/.

[2] R. Adeline, J. Cardoso, P. Darfeuil, S. Humbert, and C. Seguin. Toward a methodology for the altarica modelling of multi-physical systems. In *Proceedings of European Safety and Reliability Conference, ESREL 2010*, Rhodes (Greece), September 2010.

[3] M. AjmoneMarsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Wiley Series in Parallel Computing. John Wiley and Sons, 1994.

[4] J. Andrews and T. Moss. *Reliability and Risk Assessment*. John Wiley & Sons, 1993. ISBN 0-582-09615-4.

[5] A. Arnold, A. Griffault, G. Point, and A. Rauzy. The altarica language and its semantics. *Fundamenta Informaticae*, 34:109–124, 2000.

[6] P. Bieber, J.-P. Blanquart, G. Durrieu, D. Lesens, J. Lucotte, F. Tardy, M. Turin, C. Seguin, and E. Conquet. Integration of formal fault analysis in assert: Case studies and lessons learnt. In *Proceedings of 4th European Congress Embedded Real Time Software, ERTS 2008*, Toulouse (France), January 2008.

[7] M. Boiteau, Y. Dutuit, A. Rauzy, and J.-P. Signoret. The altarica data-flow language in use: Assessment of production availability of a multistates system. *Reliability Engineering and System Safety*, 91:747–755, 2006.

[8] J. Bowles and L. Kloul. Synthesising pepa nets from iods for performance analysis. In *Proceedings of the 1st ACM SIGMETRICS/SIGSOFT Joint WOSP/SIPEW International Conference on Performance Engineering, San Jose, California*, 2010.

[9] P. Brameret, A. Rauzy, and J. Roussel. Assessing the dependability of systems with repairable and spare components. In J. Barbet, editor, *Actes du Congrès Lambda-Mu 18*, Octobre 2012.

[10] S. Donatelli. Superposed generalised stochastic petri nets: Definition and efficient solution. In M. Silva, editor, *Proceedings of 15th International Conference on Application and Theory of Petri Nets*, 1994.

[11] S. Gilmore, J. Hillston, and L. Kloul. Pepa nets. In M. Calzarossa and E. Gelenbe, editors, *Performance Tools and Applications to Networked Systems*, volume 2965, pages 311–335. LNCS, Springer-Verlag, 2004.

[12] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. Pepa nets: a structured performance modelling formalism. *Performance Evaluation*, 54(2):79–104, 2003.

[13] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. Software performance modelling using pepa nets. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Software and Performance (WOSP'04), Redwood City, California*, 2004.

[14] J. Hillston. The nature of synchronisation. In U. Herzog and M. Rettelbach, editors, *Proceedings of 2nd Process Algebra and Performance Modelling Workshop*, November 1994.

[15] J. Hillston. Tuning systems: From composition to performance. *The Computer Journal*, 48(4):385–400, 2005.

[16] K. Jensen. *Coloured Petri Nets, Volume 1: Basic Concepts*. Springer-Verlag, 1992.

[17] R. Milner. Communicating and mobile systems: The pi-calculus. *Cambridge University Press*, 1999.

[18] T. Prosvirnova and A. Rauzy. Système de transitions gardées : formalisme pivot de modélisation pour la sûreté de fonctionnement. In J. Barbet, editor, *Actes du Congrès Lambda-Mu 18*, Octobre 2012.

[19] A. Rauzy. BDD for Reliability Studies. In K. Misra, editor, *Handbook of Performability Engineering*, pages 381–396. Elsevier, 2008. ISBN 978-1-84800-130-5.

[20] A. Rauzy. Guarded transition systems: a new states/events formalism for reliability studies. *Journal of Risk and Reliability*, 222(4):495–505, 2008.

[21] A. Rauzy. Anatomy of an efficient fault tree assessment engine. In R. Virolainen, editor, *Proceedings of International Joint Conference PSAM'11/ESREL'12*, June 2012.

[22] W. Sanders and J. Meyer. Reduced base model construction methods for stochastic activity networks. *IEEE Journal on Selected Areas in Communications*, 9(1):25?36, January 1991.

[23] R. Valk. Petri nets as token objects-an introduction to elementary object nets. In *Proc. of the 19th International Conference on Application and Theory of Petri Nets, volume 1420 of LNCS, pages 1?25, Lisbon*. Springer Verlag, 1998.

Appendix B

AltaRica and Safety Analysis Modeling Language (SAML)

Many states/transitions formalisms have been proposed in the literature to perform Safety Analyses. In this chapter, we compare two of them: Safety Analysis Modeling Language (SAML) and AltaRica. These formalisms have been developed by different communities. Their look-and-feel are thus quite different. Yet, their underlying mathematical foundations are very similar: both of them rely on state automata.

SAML was designed as a tool independent formal system specification and modeling language [46]. A SAML model is expressed in terms of finite stochastic state automata. A model may consist of more than one automata, which are all executed in discrete time with parallel composition. Besides technical systems with deterministic behavior, SAML may also denote failure models with stochastic behavior and system environments which often have non-deterministic behavior. Due to the combination of stochastic and non-deterministic specification, the semantics of a SAML model is defined as a Markov decision process.

AltaRica [7, 88] has been designed with an engineering perspective. AltaRica models are made of hierarchies of reusable components. Graphical representations are associated to components, making models visually very close to Process and Instrumentation Diagrams.

It is of interest to compare both formalisms in order to study their ability to assess the reliability of systems, to highlight their respective advantages and drawbacks and to seek for opportunities of a cross fertilization. We compare these two formalisms according to the following axes:

- the high level structural constructions;
- the underlying finite state automata;
- the representation and interpretation of time.

Comparison of Modeling Formalisms for Safety Analyses: SAML and AltaRica

Michael Lipaczewski[a], Frank Ortmeier[a], Tatiana Prosvirnova[b], Antoine Rauzy[b], Simon Struck[a]

[a]*Otto-von-Guericke University Magdeburg, Computer Systems in Engineering, Magdeburg, GERMANY*
[b]*LIX - Ecole Polytechnique, route de Saclay, 91128 Palaiseau cedex, FRANCE*

Abstract

Many states/transitions formalisms have been proposed in the literature to perform Safety Analyses. In this paper we compare two of them: SAML and AltaRica. These formalisms have been developed by different communities. Their "look-and-feel" are thus quite different. Yet, their underlying mathematical foundations are very similar: both of them rely on state automata. It is therefore of interest to study their ability to assess the reliability of systems, their respective advantages and drawbacks and to seek for opportunities of a cross fertilization.

Keywords: Model-based Safety Analysis, SAML, AltaRica

1. Introduction

Model based approach for Safety Analysis is gradually wining the trust of safety engineers but is still a wide domain of research. "Traditional" risk modeling formalisms, such as Fault Trees (FT) [1], Markov Processes, Generalized Stochastic Petri Nets (GSPN) [2], etc. are well known and widely used by safety engineers; and efficient algorithms and tools are available to study these models. However, despite of their qualities, these formalisms share a major drawback: models designed with these formalisms are far from the functional architecture of the system under study. As a consequence, models are hard to design and to maintain throughout the life cycle of systems. A small change in the specifications may require a complete revisit of the safety models, which is both resource consuming and error prone. The high-level modeling languages AltaRica [3, 4] and SAML [5] have been created to tackle this problem.

SAML was designed as a tool independent formal system specification and modeling language [5]. A SAML model is expressed in terms of finite stochastic state automata. A model may consists of more than one automata, which are all executed in discrete time with parallel composition. Besides technical systems with deterministic behavior SAML may also denote failure models with stochastic behavior and system environments which often have non-deterministic behavior. Due to the combination of stochastic and non-deterministic specification, the semantics of a SAML model is defined as Markov decision process.

AltaRica[3, 4] has been designed with engineering perspective. AltaRica models are made of hierarchies of reusable components. Graphical representations are associated to components, making models visually very close to Process and Instrumentation Diagrams. AltaRica is used as internal representation language by several Safety Analyses workshops: Cecilia OCAS (Dassault Aviation), Simfia (EADS Apsys), Safety Designer (Dassault Systèmes) and AltaRica Studio (LaBRI). AltaRica is a formal modeling language. Efficient algorithms have been developed to assess AltaRica models: compilation to fault trees, stochastic simulation, model-checking, generation of Markov chains, etc.

It is of interest to compare both formalisms in order to study their ability to assess the reliability of systems, their respective advantages and drawbacks and to seek for opportunities of a cross fertilization. These two formalisms are compared according to the following axes:

- the high-level structural constructions;
- the underlying finite state automata;
- the representation and interpretation of time.

To illustrate our comparison we use a case study: a power supply system. We present some qualitative and quantitative results obtained with both formalisms and the advantages and drawbacks of both formalisms.

The remainder of this article is organized as follows. Sections 2 and 3 introduce respectively SAML and AltaRica modeling languages. Section 4 gives an overview of the related works. Section 5 describes a case study, a power supply system that will be used to illustrate both formalisms. Section 6 presents SAML model and AltaRica model of the power supply system. Section 7 gives some qualitative and quantitative results obtained with SAML and AltaRica models. Section 8 compares both formalisms and, finally, Section 9 concludes this article.

Email addresses: frank.ortmeier@ovgu.de (Frank Ortmeier), prosvirnova@lix.polytechnique.fr (Tatiana Prosvirnova), rauzy@lix.polytechnique.fr (Antoine Rauzy), simon.struck@ovgu.de (Simon Struck)

2. SAML

2.1. The Language

This section provides a brief introduction to SAML. An extensive explanation of the semantics is out of scope of this paper. The interested reader is referred to Güdemann et al. [5]. In addition we evolved the language in the mean time. Thus we present the model in the currently most up to date version.

Semantically, a SAML model denotes a Markov Decision Process (MDP). This allows the modeling of time and value discrete systems with deterministic, probabilistic and non-deterministic aspects. For a formal definition of MDP see, e.g. [6].

From a syntactically point of view SAML consists of a set of *components*. Every component may contain additional components and/or state automaton. Every automaton is defined by one ore more state variables and a set of update rules. The state variables are bounded integer variables. The update rule consists of an activation condition in propositional logic and a set of states reachable if the activation condition is true. The set of reachable states is specified as a set of non-deterministic choices. Within each choice a probability distribution may be denoted. The initial state of the automaton is specified with the initial value of the state variables. *Constants*, *formulas* and *enums* can be used to increase the readability of the model. Formulas are named abbreviations for propositional logic expressions. Enums are primarily used to label system states with handy names.

Multiple automata are combined in terms of synchronous parallel composition. This means that all automata in the model move exactly one step at every time step.

2.2. Tools and Analysis

SAML was designed as tool-independent modeling language. Rather than implementing dedicated SAML centric analysis tools we use automatic semantic-preserving model transformations to transform SAML models into the input language of state-of-the-art verification engines. Integrating as much model checkers as possible allows the user to choose the most appropriate one for the problem at hand. So far there are transformations for NuSMV [7] and PRISM [8]. This transformations are semantic preserving and fully automatic [5]. With PRISM as an intermediate converter it is also possible to use MRMC [9] and we are currently busy with a converter to UPPAAL[1]. Note that none of the proposed analysis tools is limited to safety analysis.

We use NuSMV for symbolic model checking of SAML models. Due to the qualitative nature of NuSMV, the transformation from SAML into the NuSMV input language replaces all probability distributions with appropriate non-deterministic choices. The symbolic model checking approach allows the verification of SAML models against arbitrary CTL [10] properties. This solves questions like, “Is a certain (dangerous) state reachable” or “When ever X is true in one state, Y is true in the next state”.

For qualitative safety analysis we use the deductive cause consequence analysis (DCCA) [11] to compute all minimal cut-sets (i.e. critical failure combinations). It is a structured approach to search the space of failure combinations and uses model checking to test whether a hazardous state is reachable if a certain failure combination occurs. The DCCA approach is optimized to use only a minimal number of model-checking runs to exploit the complete search space. In addition to the minimal set of critical failure combinations, every combination is demonstrated by an example (i.e. sequence of states) leading to the hazard.

In addition to NuSMVs qualitative analysis, PRISM is a probabilistic model checker that exploits the stochastic information in the model. It can perform quantitative analysis like “How likely is it in the worse-case that a certain (dangerous) state is reached”. It can analyze any kind of pCTL [6] formula. Rather than calculating sets of failure combinations, the pDCCA calculates the overall hazard probability. The calculation is based on the occurrence probabilities of all discrete failures as well as the functional behavior of the model. In case of non-deterministic model components, the result is either a worse-case or best-case analysis.

3. AltaRica

This section gives an overview of AltaRica modeling language and of the associated assessment tools.

3.1. The Language

AltaRica is a high level modeling language dedicated to Safety Analyses. The first version of AltaRica was developed in LaBRI in ninetieth [12, 3]. A few years later, a second (Data-Flow) version has been developed to handle industrial scale models. A number of assessment tools have been developed for AltaRica such as compilers to Fault Trees, compilers to Markov chains, generators of critical sequences, stochastic simulators and model-checkers. Several Integrated Modeling and Simulation Environments use AltaRica as their internal representation language. Successful industrial applications have been reported [13, 14].

The third version (AltaRica 3.0) is still under specification. AltaRica 3.0 will be a major evolution of the language (and the processing tools). This new version integrates notions of object-oriented programming languages, such as inheritance and prototypes. It improves the reusability of components and knowledge capitalization. It adds also the ability to handle looped systems and to define acausal components. The models presented in this article are written in AltaRica 3.0.

AltaRica is an event-centric language because the primary objective of Safety and Reliability studies is to detect and quantify the most probable sequences of events (failures) leading the system from a nominal state to a degraded state (accident). In AltaRica, the behavior of components is described by means of Guarded Transition Systems [15, 16]. Guarded Transition

[1] 2013-03-21: `http://www.uppaal.org`

Systems generalize widely used formalisms such as Reliability Block Diagrams, Markov chains and Generalized Stochastic Petri nets. The state of a component is represented by variables (so-called state variables) and their values. The changes of state are possible when, and only when, an event occurs. The occurrence of an event updates the values of variables. Deterministic or stochastic delays can be associated with events in order to obtain (stochastic) timed models. Components can be assembled into hierarchies, their inputs and outputs can be connected and their transitions can be synchronized. So, an AltaRica model can be seen as a hierarchy of interconnected components that can be "flattened" into a unique Guarded Transition System.

3.2. Analysis

The semantics of a Guarded Transition System is a Kripke structure (a reachability graph) that can be interpreted as a Continuous-Time Markov Chain, under the condition that delays associated with transitions are exponentially distributed, or compiled into a Fault Tree.

A number of efficient assessment tools have been developed for Data-Flow Guarded Transition Systems, such as compilers to Fault Trees [17], compilers to Markov chains, generators of critical sequences of events, stochastic and stepwise simulators and model-checkers.

All the underlying algorithms can be extended to a general case of Guarded Transition Systems (without Data-Flow condition). Guarded Transition Systems make it possible to handle systems with instant loops and to define acausal components, i.e. components for which the input and output flows are decided at run time (e.g. electrical systems).

Fault Tree compiler. Fault trees are widely used to perform Safety Analyses and some regulation authorities require to use them to support the certification process. Since high-level modeling greatly improves the design, the sharing and the maintenance of models, it is of interest to use them to automatically generate Fault Trees. In many cases high-level models can be efficiently compiled into Fault Trees. The generated Fault Tree can be then assessed with calculation engines, such as XFTA [18], in order to calculate minimal cutsets, probabilities of failures, importance factors and other reliability indicators.

Markov chain generator. The compilation into Markov chains requires all the transitions to be either with exponential delays or immediate. Immediate transitions are just collapsed using the fact that an exponential delay with rate λ followed by an immediate transition of probability p is equivalent to a transition with an exponential delay of rate $p\lambda$. The problem of such a compilation is indeed the combinatorial explosion of the number of states and transitions.

Stepwise simulator. Stepwise simulator enables to perform an interactive step by step simulation of the model. This interactive tool can be very useful to debug models, to play different failure scenarios, etc. The stepwise simulator can be coupled with a graphical simulator as illustrated in [19]. Graphical simulation of models can be used to perform virtual experiments on systems, via models, helping to better understand the system behavior.

Stochastic simulator. Stochastic (Monte-Carlo) simulation is used when other assessment methods fail. The principle is to run many histories drawing at pseudo-random the delays of the transitions and to make statistics on these histories. Two types of observers can be defined to calculate reliability indicators: observers on formulas (e.g. the average number of times a formula takes a given value), observers on events (e.g. the average number of times an event has been fired). The only limit of stochastic simulation is the number of histories and the length of histories that are necessary to stabilize the measures.

Generator of critical sequences of events. A critical sequence is a sequence of events leading from the initial state to a critical state. In some cases, the order of occurrences of events does matter and thus the approximation consisting in extracting minimal cutsets (through a compilation of the model into a Fault Tree) is not suitable. In that case, minimal sequences can be extracted.

4. Related works

Many other high-level modeling languages for Safety Analyses have been defined. Two approaches for (high-level) Model-Based Safety Assessment can be found. The first one consists in creating extensions of high-level modeling languages used in other domains. The second approach consists in defining domain specific languages, dedicated to Safety Analyses. In this section we will cite some of them.

In the first category, we can find [20] who added an Error Model annex to AADL specifications, the modeling formalism for embedded real-time systems.

In the same way the HiP-HOPS workbench [21] enables the addition of reliability data to models imported from different modeling tools: Matlab/SIMULINK, Eclipse-based UML tools, etc., and then to automatically generate Fault Trees, FMEA tables, Temporal Fault Trees [22] and also to perform architecture optimisation [23] and SIL allocation [24].

Similarly, translations have been defined from specialized UML or SysML models to Fault Trees or Petri nets (see [25] or [26] for example). In [27], functional design phase, using SysML, is combined with commonly used reliability techniques (i.e. FMEA and construction of AltaRica Data-Flow models).

In the second category, we can find Figaro [28], developed by EDF R&D. It is a textual modeling language dedicated to dependability assessment of complex systems. It combines object-orientation languages features, such as inheritance, and first order production rules (interaction and occurrence rules). It is used as a description language to create knowledge bases for the workbench KB3 [29], to automatically perform systems dependability assessment: Monte-Carlo simulation, Markov Chain generation, quantification and generation of critical sequences, etc.

Component	Failure Rate	On-Demand Probability
$Grid$	$10^{-4}\ 1/h$	-
TR_1	$10^{-4}\ 1/h$	-
TR_2	$10^{-4}\ 1/h$	-
SW_1	-	10^{-3}
SW_2	-	10^{-3}
SW_3	-	10^{-3}
D	$10^{-4}\ 1/h$	10^{-3}

Table 1: Specification of the component failures

5. The Case-Study

The case study comprises of a power supply system. We adopted the case study from [30]. The model is depicted as block diagram in Figure 1. It features redundant supply lanes to avoid total system failure. In normal mode, the energy is provided from the grid via the first transformer (TR_1) and the first switch (SW_1). In case of a failure, the switch SW_2 is closed and the energy is provided via the second transformer (TR_2). If the grid or the second lane fails, the diesel engine (D) is started and the switch SW_3 is closed.

All components of the system may fail. The transformers are modeled with a per time failure. All three switches are modeled with an on-demand failure. This means that the switches can only fail to close in the instant moment where they are requested to close. Only the diesel engine is modeled with a failure rate and an on-demand failure. The failure rates and on-demand failure probabilities are denoted in Table 1. For all components, a repair rate of $10^{-1}1/h$ is specified.

The three lanes are used according to their priority, where the first lane (TR_1 and SW_1) has the highest and the third lane (D and SW_3) has the least priority. This means that when ever a component with a higher priority is repaired the according lane is used immediately after repairing.

As the case study is from the safety analysis domain, we analyze the reachability of hazardous system states. A hazard in this case is if the system fails to provide power to the *Busbar*.

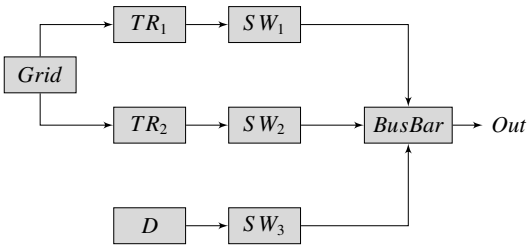


Figure 1: Block diagram of the power supply system. The arcs denote the energy flow, the blocks denote system components.

6. Modeling the System

6.1. SAML Model

Our SAML model consists of 13 modules. Eight of them are dedicated to the error modes of the components. The transformers have no internal state, so that their complete behavior is already covered by the failure modules. Four modules are reserved for the state of the three switches (i.e. open or closed) and the diesel engine (i.e. idle, running). The Busbar has no internal state, neither can it fail. This leaves one remaining module, which we explain later on. Besides the modules, our model heavily exploits formulas, denoting if a component is demanded or if certain lanes are in fail state. Constants are used for the failure and repair probabilities.

An extract of the complete SAML model is listed in Figure 2. Note that due to page limitations we listed only a minimal summary that shows the principal of the model. The formulas state elementary propositions and are mainly used to increase the maintainability and readability of the model. As there are many similar propositions for the three lanes or redundant components only some of them are listed in Figure 2.

Following the formulas, Figure 2 lists three modules: $sw1$, $trafo1\_err$ and $sw1\_demand\_err$. These modules were chosen, because they all three are characteristic for different types of modules. The $sw1$ module represents functional behavior of the system. It has two states (representing opened and closed). It closes if is_l1_demand evaluates to true and if the switch does not suffer from on-demand error ($is\_sw1\_demand\_fail$).

The module $trafo\_err$ represents a repairable (transient) per-time failure. It has two states which describe if the failure occurs or not. If in the operational state, it turns to failure state with a probability of f_trafo1. Otherwise it stays in the operational state. The behavior in the failed state is similar, but with a different probability.

The third module ($sw1\_demand\_err$) covers an on-demand failure automaton. This is slightly more complicated than the per-time failure. This remains from the following situation. The on-demand failure automaton is only allowed to change its state if the according component is demanded. At the same time the component reacts on the demand. Due to parallel composition and discrete time modeling the component in question and the failure automaton change at exactly the same time. Thus the component can only react on the changing of the failure automaton one step after the demand. The solution is to move the failure automation exactly once some time before it is required. This requires the model to have a zeroed time step where the on-demand failure modules are initialized. That is why the on-demand failure module has three states (initial, failure and operational). A detailed explanation of failure modeling in SAML is provided in [5]. A more detailed discussion about the initial step can be found in [31]

To comply with the discrete time semantics of SAML we assigned a time of 1 minute to every step ($\Delta t = 1min$). This affects the conversion from failure rates in the system specification to per-step probabilities in the model.

The formula is_no_power denotes if the system fails to provide power to the BusBar. However it is not sufficient to analyze whether the system may reach such a state. Because SAML uses a discrete timed semantics, an information needs some time to propagate throw the component structure. For the presented case study, the redundant power lanes can only react after the system has failed. Thus a single powerless state is not

```
component main

constant double f_trafo1 := 10E-12; // 1/min
constant double r_trafo1 := 6E-12; // 1/min
constant double f_sw1_demand := 10E-12; // 1/min
constant double r_sw1 := 6E-12; // 1/min
[..]

formula is_l1_demand := !is_l1_fail
formula is_sw1_demand :=
          (is_l1_demand & is_sw1_open);
formula is_trafo1_fail := trafo1_e = 1;
formula is_sw1_demand_fail := sw1_demand_e = 1;
formula is_l1_fail :=
          (is_grid_fail|is_sw1_open|is_trafo1_fail);
formula is_no_power :=
          (is_l1_fail & is_l2_fail & is_l3_fail);
[..]

component sw1
  enum SW_STATE := [OPEN, CLOSE];
  sw1_s : SW_STATE init CLOSE; // OPEN, CLOSE

  sw1_s=OPEN & (!is_sw1_demand | is_sw1_demand_fail) ->
      choice:(1:(sw1_s'= OPEN));
  sw1_s=OPEN & is_sw1_demand & !is_sw1_demand_fail ->
      choice:(1:(sw1_s'= CLOSE));
  sw1_s=CLOSE & !is_l1_demand -> choice:(1:(sw1_s'= OPEN));
  sw1_s=CLOSE & is_l1_demand -> choice:(1:(sw1_s'= CLOSE));
endcomponent

component trafo1_err
  enum ERR_STATE := [OK, FAIL];
  trafo1_e : ERR_STATE init OK; // OK, FAIL

  trafo1_e=OK -> choice:(f_trafo1:(trafo1_e'=FAIL) +
        (1-f_trafo1):(trafo1_e'=OK));
  trafo1_e=FAIL -> choice:(r_trafo1:(trafo1_e'=OK) +
        (1-r_trafo1):(trafo1_e'=FAIL));
endcomponent

component sw1_demand_err
  enum DEMAND_STATE := [OK, ERR, INI];
  sw1_demand_e : DEMAND_STATE init INI; // OK, ERR, INIT

  sw1_demand_e=INI -> f_sw1_demand:(s'=DE_E.ERR) +
      (1-f_sw1_demand):(s'=DE_E.OK);
  sw1_demand_e=OK & !is_sw1_demand ->
      choice:(1:(sw1_demand_e'=0));
  sw1_demand_e=OK & is_sw1_demand ->
      choice:( f_sw1_demand:(sw1_demand_e'=ERR) +
        (1-f_sw1_demand):(sw1_demand_e'=OK));
  sw1_demand_e=ERR -> choice:(r_sw1:(sw1_demand_e'=OK) +
        (1-r_sw1):(sw1_demand_e'=ERR));
endcomponent
[..]

endcomponent
```

Figure 2: Extract of the SAML model

considered a hazardous state. To compensate this effect, a last module was introduced. This module counts the subsequent steps where the system fails to provide power. A failure occurs whenever the counter reaches two. The counter has the state variable obs_s so that the hazard is defined as $H := obs\_s = 2$.

6.2. AltaRica Model

The power supply system is composed of 4 types of components: a grid, a transformer, a switch and a diesel engine. As discussed in Section 5, we shall consider the following failure modes:

- a grid and a transformer can only fail in operation (stochastic exponentially distributed event with a failure rate λ);
- a switch can fail on demand (with a probability γ) or be turned on successfully;
- a diesel engine can either fail in operation or on demand.

The behavior of these components can be represented by different modeling patterns of Guarded Transition Systems, pictured Figure 3. The AltaRica code corresponding to the finite state machine of a spare component (representing a diesel engine) is given Figure 4.

6.2.1. States

The internal state of the `SpareComponent` is represented by means of the state variable `state`. `state` takes its values in the domain `SpareComponentState` declared upfront. The initial values of state variables are specified by means of the attribute `init`.

6.2.2. Events

The state of the component changes under the occurrence of an event. Events are introduced with the keyword `event`. A delay is associated with each event by means of the attribute `delay`. Delays of events `failure` and `repair` are random variables exponentially distributed with respective rates `lambda` and `mu`. Events `start` and `failureOnDemand` are instantaneous (their delay is 0). Both are fireable when the component is `OFF`. `start` has the probability `1 - gamma` to be fired while `failureOnDemand` has a probability `gamma` to be fired in this state. This probability is given through the attribute `expectation`. The expectation of the event e is used to determine the probability that the transition labeled with e is fired in case of several transitions are fireable at the same date. When transitions labeled with $e_1, e_2, \ldots, e_k$ are scheduled at the same date, the probability $p(e_i)$ to fire the transition labeled with e_i $(1 \leq i \leq k)$ is defined as follows:

$$p(e_i) = \frac{expectation(e_i)}{\sum_{1 \leq j \leq k} expectation(e_j)}$$

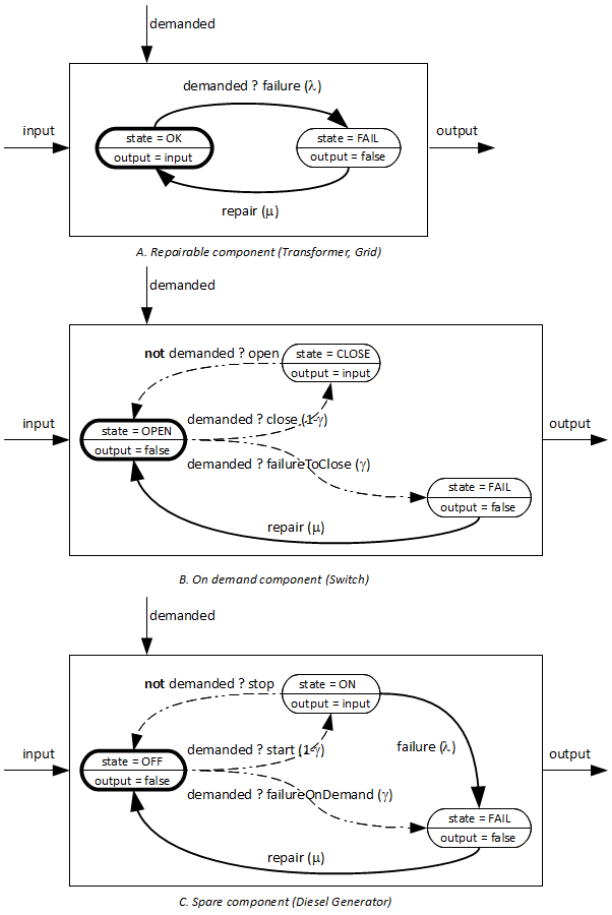


Figure 3: Patterns of Guarded Transition Systems.

```
domain SpareComponentState { ON, OFF, FAIL }

class SpareComponent
  SpareComponentState state (init = OFF);
  Boolean demanded (reset = false);
  Boolean input (reset = false);
  Boolean output (reset = false);
  Boolean failed (reset = false);
  event start (delay = 0, expectation = 1 - gamma);
  event failureOnDemand (delay = 0,
           expectation = gamma);
  event failure (delay = exponential(lambda));
  event repair (delay = exponential(mu));
  event stop (delay = 0);
  parameter Real gamma = 10e-3;
  parameter Real lambda = 10e-4;
  parameter Real mu = 10e-1;
transition
  start:  state==OFF and demanded → state := ON;
  failureOnDemand:state==OFF and demanded →
           state := FAILED;
  failure:  state==ON → state := FAIL;
  repair:  state==FAIL → state := OFF;
  stop:  state==ON and not demanded → state := OFF;
assertion
  output := if state==ON then input else false;
  failed := (state==FAIL);
end
```

Figure 4: The AltaRica code for the Finite State Automaton modeling a spare component.

6.2.3. *Transitions*

A transition is a triple $\langle e, G, P \rangle$, also denoted $e : G \rightarrow P$, where e is an event, G is a Boolean expression, so-called the guard (or the pre-condition) of the transition, P is an instruction, so-called the action (or the post-condition) of the transition. Transitions are described in the clause `transition`. If the state of the component is `OFF`, then two transitions are fireable: the transition labeled with the event `start` and the transition labeled with the event `failureOnDemand`. These transitions are deterministic and instantaneous because they are associated with a delay 0. The transition labeled by `failureOnDemand` has the probability to be fired `gamma` and the transition labeled by `start` has the probability to be fired `1-gamma`. If the transition labeled by `failureOnDemand` is fired, then its action is executed: `state` is switched to `FAIL`. In the Figure 3 instantaneous transitions are marked with dashed lines. Transitions `failure` and `repair` are timed and stochastic. They obey typically exponential distributions. In the Figure 3 timed transitions are marked with plain lines. If the state of the component is `ON` and the delay drawn for the transition `failure` is the shortest, then this transition is fired.

6.2.4. *Parameters*

Parameters are constant values that come with the definition of the AltaRica class. When a class is instantiated, their values may be changed. In the model above, there are three parameters `gamma`, `lambda` and `mu` that define respectively the probability of failure on demand and the failure and repair rates.

6.2.5. Flow variables and assertions

Variables `demanded`, `input`, `output` are Boolean flow variables. The variable `demanded` is used to implement the command, i.e. to tell when to turn on and off the component. The variables `input` and `output` represent the flow circulating through the component, and in this case it is the electrical power. From a syntactic viewpoint, flow variables are introduced (and distinguished from state variables) by means of the attribute `reset`. Conversely to state variables, that are initialized at the beginning of a run and then modified through actions of transitions, the value of flow variables are recalculated after each transition firing. This recalculation is performed by means of assertions. Assertions are instructions just as actions of transitions. The difference stands in that actions of transitions assign state variables only while assertions assign flow variables only. Moreover, each component has a unique assertion that is applied after each transition firing.

Flow variables and assertions are used to model information flows circulating through a system. They may represent physical connections between components, control commands, fluid circulation, electric power, etc. They offer an easy and elegant way to express dependencies on external factors.

The AltaRica code for the `SpareComponent` and for the other patterns (`OnDemandComponent` and `RepairableComponent`) are quite similar.

6.2.6. Composition

Now we can consider the model for the whole power supply system. AltaRica 3.0 is an object-oriented modeling language. Therefore, the AltaRica class that describes the power supply system embeds an instance of the class `SpareComponent` describing the diesel engine `D`, three instances of the class `OnDemandComponent` representing the switches `SW1, SW2, SW3`, and three instances of the class `RepairableComponent` describing the grid `Grid` and the transformers `TR1, TR2`, as illustrated Figure 5.

When the lane 1 is failed, the switch of the second lane `SW2` is attempted to turn on. If it fails then the diesel generator `D` is attempted to start and the switch `SW3` is attempted to turn on. These rules are expressed in the assertion of the class `PowerSupplySystem`.

```
class PowerSupplySystem
  RepairableComponent Grid, TR1, TR2;
  SpareComponent D;
  OnDemandComponent SW1(s.init = CLOSE);
  OnDemandComponent SW2, SW3;
  Boolean lane1_failed(reset = false);
  Boolean lane2_failed(reset = false);
  Boolean lane3_failed(reset = false);
  observer Boolean failed = lane1_failed and
           lane2_failed and lane3_failed;
assertion
  lane1_failed := Grid.failed or TR1.failed
           or SW1.failed;
  lane2_failed := Grid.failed or TR2.failed
           or SW2.failed;
  lane3_failed := D.failed or SW3.failed;
  Grid.demanded := not lane1_failed or
           not lane2_failed;
  TR1.demanded := not lane1_failed;
  SW1.demanded := TR1.demanded;
  TR2.demanded := lane1_failed and
           not lane2_failed;
  SW2.demanded := TR2.demanded;
  D.demanded := lane1_failed and lane2_failed
           and not lane2_failed;
  SW3.demanded := D.demanded;
  Grid.input := true;
  D.input := true;
  TR1.input := Grid.output;
  SW1.input := TR1.output;
  TR2.input := Grid.output;
  SW2.input := TR2.output;
  SW3.input := D.output;
end;
```

Figure 5: The AltaRica code for the whole system.

6.2.7. 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 the assessment tools. Observers are updated after each transition firing.

In the AltaRica code of the class `PowerSupplySystem` we declared an observer `failed` that detects if the system is in the hazardous state (when all the lanes are failed and, therefore, the system cannot supply power to the *Busbar*).

7. Analyzing the System

7.1. SAML

We performed a DCCA on the SAML model, which lead to the following minimal cut-sets:

- $\Gamma_1 = \{grid\_e, sw3\_demand\_e\}$
- $\Gamma_2 = \{grid\_e, diesel\_demand\_e\}$
- $\Gamma_3 = \{grid\_e, diesel\_e\}$
- $\Gamma_4 = \{trafo1\_e, sw2\_demand\_e, sw3\_demand\_e\}$
- $\Gamma_5 = \{trafo1\_e, trafo2\_e, sw3\_demand\_e\}$
- $\Gamma_6 = \{trafo1\_e, diesel\_demand\_e, sw2\_demand\_e\}$
- $\Gamma_7 = \{trafo1\_e, diesel\_e, sw2\_demand\_e\}$
- $\Gamma_8 = \{trafo1\_e, trafo2\_e, diesel\_demand\_e\}$
- $\Gamma_9 = \{trafo1\_e, trafo2\_e, diesel\_e\}$

In addition to DCCA we use PRISM to perform a probabilistic deductive cause consequence analysis (pDCCA) [5]. To analyze the overall hazard probability we used the pCTL property $P_{max=?}[trueU <= nobs_s = 2]$ where n denotes the number of steps to analyze. According to $\Delta t = 1min$ one hour leads to $n = 60$, two hours to $n = 600$ and so forth. The results of the PRISM based analysis are listed in Table 2.

1h	10h	100h	10000h
1.00e-9	1.36e-6	2.24e-5	2.47e-3

Table 2: Hazard Probability

7.2. AltaRica

In order to perform different types of analyses, the AltaRica model given in Figure 5 is first of all "flattened" into a unique Guarded Transition System (GTS). For the obtained GTS it is possible to generate a Reachability graph that can be explored in order to compute reliability indicators. In this article we focus on two types of model analyses:

- the generation of critical sequences of events;
- the compilation into a Markov chain in order to compute the system unreliability $P[T < t]$, the probability that the system fails (the property $failed == true$ becomes verified) before the time t.

The generated critical sequences are:

1. G.failure SW3.start D.failureOnDemand
2. G.failure SW3.start D.start, D.failure
3. G.failure SW3.failureOnDemand
4. TR1.failure SW2.start TR2.failure SW3.start D.failureOnDemand
5. TR1.failure SW2.start TR2.failure SW3.start D.start D.failure
6. TR1.failure SW2.start TR2.failure SW3.failureOnDemand
7. TR1.failure SW2.failureOnDemand SW3.start D.failureOnDemand
8. TR1.failure SW2.failureOnDemand SW3.start D.start D.failure
9. TR1.failure SW2.failureOnDemand SW3.failureOnDemand

The generated Markov graph contains 72 states and 248 transitions. The obtained results are summarized in Table 3.

1h	10h	100h	10000h
2.06e-5	2.11e-4	2.12e-3	1.91e-1

Table 3: Unreliability

8. Comparison and Evaluation

To compare Safety modeling formalisms we have established several comparison criteria. They are discussed below.

Event based. The goal of Safety Analyses is to determine the most probable failure scenarios, i.e. sequences of events leading from the nominal state to a failure/hazardous state. There are potentially different types of events: stochastic, instantaneous, timed deterministic, etc.

AltaRica 3.0 is an event based modeling language: it is possible to explicitly name events and associate them to transitions. Events can be stochastic (e.g. events $failure$ and $repair$ in the model Figure 4) and instantaneous (e.g. events $failureOnDemand$, $start$ and $stop$ in the model Figure 4).

In the sense of SAML, all continuous functions are sampled and processed in discrete time steps. Though, the subsequent state of a SAML model is solely based on the current state and the decision of non-deterministic and probabilistic choices. Events that occur in between two time steps, are cumulatively processed when forming the next state.

Composition. Models of systems should be obtained by composing models of subsystems. States of the system should be given in a implicit way to avoid the user to enumerate all of them and to allow approximations, based on the most probable scenarios/states.

AltaRica 3.0 and SAML are both compositional and represent implicitly state graphs of modeled systems. A SAML model is composed of several components that are all executed in parallel. Recently, templates have been added to SAML: it allows to create component instances based on a pattern. They not only greatly improve the reusability of models but allow the convenient modeling of equal components. In AltaRica 3.0 each system component is represented by a class; a system model is obtained by instantiating previously defined classes, connecting their inputs and outputs and synchronizing their events (see, for example, the model of the case study Figure 5). Unlike SAML, in AltaRica only one transition can be fired at a time.

Hierarchy. Models of systems should be obtained by composing models of subsystems or different views of the system into hierarchies.

AltaRica 3.0 integrates notions of object-oriented programming languages such as inheritance and prototypes. It offers constructs to structure models into hierarchies of reusable components. An AltaRica 3.0 model can be seen as a hierarchy of interconnected components (see, for example, the model of the case study Figure 5).

Recently, the notion of nested component has been added to SAML. A SAML model can be represented as a hierarchy of nested components.

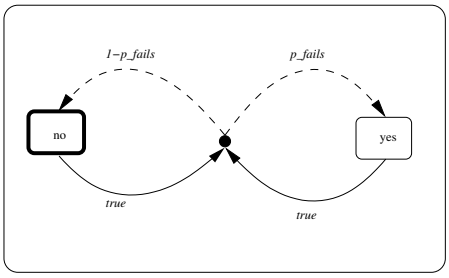


Figure 6: Graphical representation of transient failure automaton.

Remote interactions. It should be possible to describe easily remote interactions between components, i.e. flows of matter or information circulating through the system (without enumerating them explicitly).

In AltaRica 3.0 remote interactions are represented by flow variables and assertions. In the example Figure 5 the assertion calculates system flow variables. The principle is explained in Section 6.2. The assertion is recalculated after each transition firing. Remote interactions can be also expressed by synchronizations of events. Examples can be found in [32].

In SAML the remote interactions can be represented by means of shared variables. In general, every state variable in SAML is globally readable so the current state of one component is implicitly distributed to all others. To increase readability of the model, we use formulas to assign names to relevant (sets of) states. For example, in the model, given Figure 2 the formulas are used for that purpose.

Graphical representation. Graphical representation of models has its own interest. One should be able to represent graphically models, at least partly, for communication and animation purposes.

It is not possible to have a unique graphical representation of an AltaRica 3.0 model. At least three different graphical views can be used to represent it:

1. Representations, like Process & Instrumentation Diagrams or Block Diagrams (see Figure 1), can be used to capture the hierarchy, the connections between components and the circulating flows of AltaRica 3.0 models.
2. State diagrams, like those given Figure 3, can represent the internal behavior of each AltaRica 3.0 class.
3. Thus events in AltaRica 3.0 can be synchronized, diagrams, like UML sequence diagrams, can be used to represent synchronizations.

Each of these graphical representations gives a partial view of an AltaRica 3.0 model.

SAML naturally maps to state charts. An example for a transient failure module is given in Figure 6. The combination of non-deterministic and probabilistic choices leads to a slightly more extensive notation. Every probability distribution is denoted by dashed arrows leaving the same black dot. Possible non-deterministic choices can then be expressed by multiple arrows with the same guard and leaving the same state.

	SAML	AltaRica 3.0
Event based	No	Yes
Composition	Templates	Object-oriented
Hierarchy	Nested components	Object-oriented
Remote interactions	Shared variables	Flow variables & assertion, synchronizations
Graphical	State charts	State, Sequence & Block diagrams
Assessment tools	Model-checking, probabilistic model-checking, stochastic simulation	Compilation to Fault Trees & Markov graphs, stochastic & stepwise simulation
Time	Discrete	Triggered by events

Table 4: Comparison of Safety formalisms: SAML and AltaRica 3.0.

Available assessment tools. Prototypes of a set of assessment tools for the new version of AltaRica are currently developed. They include a Fault Tree compiler, a Markov chain generator, a stepwise and a stochastic simulators. These assessment tools will be distributed under a free licence.

At the current state, there exist three assessment tools for SAML. For one there are the NuSMV and Prism model checkers (described in Section 2). In addition the VECS[2] [33] tool contains a step by step simulator. Besides the existing tools, the framework is designed in a flexible way, so that additional tools can be integrated easily.

Interpretation of time. Time in AltaRica and SAML is interpreted differently. SAML is synchronous. It uses a discrete time model. The state of all automaton in the model is updated only at discrete time steps. The successive state solely depends on the current state.

The time in AltaRica is triggered by events. It is an intermediate model between discrete and continuous time. A delay is associated with each event. It can be deterministic or stochastic and may depend on the state. When the transition labeled with the event gets fireable at time t, a delay d is calculated, and the transition is actually fired at time $t + d$ if it stays fireable from t to $t + d$. The semantics of AltaRica model is a Kripke structure (a reachability graph) that can be interpreted as a continuous-time Markov chain, under the condition that delays associated with transitions are exponentially distributed.

9. Conclusion and Outlook

In this paper we compared SAML and AltaRica. Both are formal modeling languages for Model-Based Safety Analysis. On the syntactical level the set of language constructs in SAML appeared to be smaller than the one in AltaRica. On the one hand this simplifies the models but on the other hand also makes it more difficult to express large and/or complex systems.

[2]The tool was formerly named S3E.

On the semantic level the two languages chose fundamentally different approaches. SAML uses a discrete time model with equidistant time steps. AltaRica is based on continuous time with discrete events. In practice this means that in SAML all automata perform exactly one transition at the same time. In AltaRica only one transition is fired at one time.

In future work we will evaluate the conversion between AltaRica and SAML. Even though not trivial, an automatic conversion between the two languages extends their set of available analysis tools. The main challenge for such a transformation is for sure the different time-model in the two languages.

For SAML we are currently busy with the evaluation of a data-flow based modeling approach like in AltaRica. The case-study we used in this paper mostly consists of data-flow, which was rather difficult to express in SAML. Never the less, the SAML language should remain as simple as possible.

References

[1] J. Andrews, T. Moss, Reliability and Risk Assessment, John Wiley & Sons, 1993, iSBN 0-582-09615-4.

[2] M. AjmoneMarsan, G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, Modelling with Generalized Stochastic Petri Nets, Wiley Series in Parallel Computing, John Wiley and Sons, 1994.

[3] A. Arnold, A. Griffault, G. Point, A. Rauzy, The altarica language and its semantics, Fundamenta Informaticae 34 (2000) 109–124.

[4] M. Boiteau, Y. Dutuit, A. Rauzy, J.-P. Signoret, The altarica data-flow language in use: Assessment of production availability of a multistates system, Reliability Engineering and System Safety 91 (2006) 747–755.

[5] M. Güdemann, F. Ortmeier, A framework for qualitative and quantitative model-based safety analysis, in: Proceedings of the 12^{th} High Assurance System Engineering Symposium (HASE 2010), 2010, pp. 132–141.

[6] L. de Alfaro, M. Faella, T. A. Henzinger, R. Majumdar, M. Stoelinga, Model checking discounted temporal properties, Theoretical Computer Science 345 (2005) 139–170.

[7] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking, in: Proceedings of the 14^{th} International Conference on Computer Aided Verification (CAV 2002), Vol. 2404 of LNCS, Springer, 2002.

[8] M. Kwiatkowska, G. Norman, D. Parker, Probabilistic symbolic model checking with PRISM: A hybrid approach, in: Proceedings of the 8^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002), Vol. 2280 of LNCS, Springer, 2002.

[9] J.-P. Katoen, M. Khattri, I. Zapreev, A Markov reward model checker, in: Proceedings of the 2^{nd} International Conference on Quantitative Evaluation of Systems (QEST 2005), IEEE Computer Society, 2005.

[10] E. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 2000.

[11] M. Güdemann, F. Ortmeier, W. Reif, Computing ordered minimal critical sets, in: E. Schnieder, G. Tarnai (Eds.), Proceedings of the 7^{th} Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2008), 2008.

[12] G. Point, A. Rauzy, AltaRica: Constraint automata as a description language, Journal Européen des Systèmes Automatisés 33 (8–9) (1999) 1033–1052.

[13] R. Bernard, J.-J. Aubert, P. Bieber, C. Merlini, S. Metge, Experiments in model-based safety analysis: flight controls, in: Proceedings of IFAC workshop on Dependable Control of Discrete Systems, Cachan, 2007.

[14] R. Bernard, S. Metge, F. Pouzolz, P. Bieber, A. Griffault, M. Zeitoun, Altarica refinement for heterogeneous granularity model analysis, in: Actes du congrs Lambda-Mu16, Avignon, 2008.

[15] A. Rauzy, Guarded transition systems: a new states/events formalism for reliability studies, Journal of Risk and Reliability 222 (4) (2008) 495–505.

[16] T. Prosvirnova, A. Rauzy, Guarded transition systems: Pivot modelling formalism for safety analysis, in: J. Barbet (Ed.), Actes du Congrès Lambda-Mu 18, 2012.

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

[18] A. Rauzy, Anatomy of an efficient fault tree assessment engine, in: R. Virolainen (Ed.), Proceedings of International Joint Conference PSAM'11/ESREL'12, 2012.

[19] B. Perrot, T. Prosvirnova, A. Rauzy, J.-P. S. d'Izarn, R. Schoening, Expériences de couplages de modèles AltaRica avec des interfaces métiers, in: E. Fadier (Ed.), Actes du congrès LambdaMu'17 (actes électroniques), IMdR, 2010.

[20] P. Feiler, A. Rugina, Dependability modeling with the architecture analysis & design language (aadl), Tech. rep., Carnegie Mellon University (2007).

[21] A. Pasquini, Y. Papadopoulos, J. McDermid, Hierarchically performed hazard origin and propagation studies, Computer Safety, Reliability and Security 1698 of LNCS (1999) 688–688.

[22] M. Walker, Y.Papadopoulos, Qualitative temporal analysis: Towards a full implementation of the fault tree handbook, Control Engineering Practice 17 (2009) 1115 – 1125.

[23] Y. Papadopoulos, M. Walker, D. Parker, E. Rüde, R. Hamann, A. Uhlig, U. Grätz, R. Lien, Engineering failure analysis and design optimisation with HiP-HOPS, Engineering Failure Analysis 18 (2) (2011) 590 – 608, the Fourth International Conference on Engineering Failure Analysis Part 1.

[24] Y. Papadopoulos, M. Walker, M.-O. Reiser, M. Weber, D. Chen, M. Törngren, D. Servat, A. Abele, F. Stappert, H. Lonn, L. Berntsson, R. Johansson, F. Tagliabo, S. Torchiaro, A. Sandberg, Automatic allocation of safety integrity levels, in: Proceedings of the 1st Workshop on Critical Automotive Applications: Robustness & Safety, CARS '10, ACM, New York, USA, 2010, pp. 7–10.

[25] J. Xiang, K. Yanoo, Y. Maeno, K. Tadano, Automatic synthesis of static fault trees from system models, in: Conference on Secure Software Integration and Reliability Improvement, 2011, p. 127136.

[26] S. Bernardi, S. Donatelli, J. Merseguer, From uml sequence diagrams and statecharts to analyzable petri net models, in: In Proceedings of the Third International Workshop on Software on Performance, 2002.

[27] P. David, V. Idasiak, F. Kratz, Reliability study of complex physical systems using sysml, Reliability Engineering and System Safety (2010) 431–450.

[28] M. Bouissou, H. Bouhadana, M. Bannelier, N. Villatte, Knowledge modelling and reliability processing: presentation of the figaro modelling language and associated tools, in: Proceedings of Safecomp'91, 1991.

[29] M. Bouissou, Automated dependability analysis of complex systems with the kb3 workbench: the experience of edf r&d, in: Proceedings of the International Conference on Energy and Environment, 2005.

[30] M. Bouissou, J.-L. Bon, A new formalism that combines advantages of fault-trees and markov models: Boolean logic driven markov processes, Reliability Engineering & System Safety 82 (2) (2003) 149 – 163. doi:DOI: 10.1016/S0951-8320(03)00143-1.
URL `http://www.sciencedirect.com/science/article/B6V4T-49DFH1M-1/2/bd15510dc655e0bbc55f3e5758bdeb42`

[31] M. Güdemann, Qualitative and Quantitative Formal Model-Based Safety Analysis, Ph.D. thesis, Otto-von-Guericke-Universitä Magdeburg (2011).
URL `http://nbn-resolving.de/urn:nbn:de:gbv:ma9:1-385`

[32] L. Kloul, T. Prosvirnova, , A. Rauzy, Modeling systems with mobile components: a comparison between altarica and pepa nets, Journal of Risk and Reliability 227 (6) (2013) 599–613.

[33] M. Lipaczewski, S. Struck, F. Ortmeier, Saml goes eclipse - combining model-based safety analysis and high-level editor support, in: Proceedings of the 2nd International Workshop on Developing Tools as Plug-Ins (TOPI), IEEE, 2012, pp. 67–72.

Appendix C

Graphical representation and animation of models

One of the main contributions of the Model-Based approach for Safety Assessment is the ability to graphically simulate incident or accident scenarios. As a consequence, incident or accident scenarios can be visualized and discussed.

Graphical simulators are integrated into all the Integrated Modeling and Simulation Environments for AltaRica Data-Flow: Cecilia OCAS (Dassault Aviation), Safety Designer (Dassault Systemes) and Simfia (EADS Apsys). These environments make it possible to create, to edit and to simulate models graphically. The same graphical representation is used to create and to animate the model.

In this chapter we propose a different approach: distinguish the modeling environment (i.e. graphical representation/creation/edition of models) from the simulation/animation environment (i.e. graphical animation of models) as illustrated in reference [79]. We use a Model-Based approach for graphical animation of models, which consists in the definition of a high level modeling language for graphical animation of models.

In the following article, we present GraphXica - a Domain Specific Language for graphical animation of models. GraphXica has the same structural constructs as AltaRica 3.0 (see Chapter 3). It enables to describe graphical primitives (lines, rectangles, circles, etc.) and their animations (color change, scale, rotation, translation, etc.) according to the value of external variables. The value of variables can be given by the user or can be provided by a simulator.

The following prototype has been developed: the stepwise simulator of AltaRica 3.0 described in Chapter 5 has been coupled with GraphXica Displayer. The communication between the tools is performed via a text file. The stepwise simulator of AltaRica 3.0 writes the value of all variables in the file after each simulation step (typically the firing of a transition). The GraphXica Displayer reads the value of variables every two seconds (this parameter can be defined by the user) and refreshes the graphical representation according to the described animation rules and the value of variables.

GraphXica: a Language for Graphical Animation of Models

T. Prosvirnova, M. Batteux, A. Maarouf & A. Rauzy
LIX
Ecole Polytechnique, Palaiseau, France

ABSTRACT: The objective of this article is to present GraphXica – a Domain Specific Language (DSL) for graphical animation of models. GraphXica enables to describe graphical representations of models and their animations. Given a graphical representation and animation description of the model, different kinds of Graphical User Interfaces (GUIs) can be generated to simulate it, for example a web based interface, a Java interface, etc.
This work is a part of AltaRica 3.0 project, which aims to propose a set of authoring, simulation and assessment tools to perform Model-Based Safety Analyses. The new version of AltaRica modeling language is in the heart of the project. It is a textual language but graphical representations can be easily associated to textual models. GraphXica can be used to define graphical representations and animations of AltaRica 3.0 models. Then a GUI can be generated to animate these models. Coupled with a stepwise simulator, it enables to perform virtual experiments on systems, via models.
GraphXica is a generic DSL and can be used to describe graphical representations and animations of any kind of models and data.

1 INTRODUCTION

The Model-Based approach for safety and reliability analysis is gradually wining the trust of engineers but is still an active domain of research. Safety engineers master "traditional" risk modeling formalisms, such as "Failure Mode, Effects and Criticality Analysis" (FMECA), Fault Trees (FT), Event Trees (ET), Markov Processes. Efficient algorithms and tools are available. However, despite of their qualities, these formalisms share a major drawback: models are far from the specifications of the systems under study. As a consequence, models are hard to design and to maintain throughout the life cycle of systems. A small change in the specifications may require revisiting completely safety models, which is both resource consuming and error prone.

The high-level modeling language AltaRica Data-Flow (Rauzy 2002, Boiteau et al. 2006) has been created to tackle this problem. AltaRica Data-Flow models are made of hierarchies of reusable components. Graphical representations are associated with components, making models visually very close to Process and Instrumentation Diagrams. It is in the core of several Safety Analysis workshops and several successful industrial experiments have been held using AltaRica Data-Flow (Bernard et al. 2007, Bieber et al. 2008).

However, more than ten years of experience showed that both the language and the assessment tools can be improved. AltaRica 3.0 is an entirely new version of the language. Its underlying mathematical model – Guarded Transition Systems (Rauzy 2008, Prosvirnova and Rauzy 2012) – makes it possible to design acausal components and to handle looped systems. The development of a complete set of freeware authoring, simulation and assessment tools is planed, so to make them available to a wide audience.

The success AltaRica, partially, comes from the fact that graphical representations can be easily associated to textual models. Thus, models can be graphically animated. The incident or accident scenarios can be visualized and discussed. In a word, virtual experiments on systems can be performed using these models.

As a part of AltaRica 3.0 project our team works on the graphical representation and simulation of AltaRica 3.0 models. The goal of this communication is to present a Model-Based approach for 2D-3D event driven simulation. This approach consists in the definition of a Domain Specific Language (DSL) for graphical animation of models. This language enables to describe graphical 2D-3D representations of models and their animations. Then, it can be used to generate a graphical user interface (GUI) to animate models. The major advantage of this approach is that given

a graphical representation and an animation description of the model, different kinds of GUIs can be generated to simulate it, for example, a web interface or a java interface. The generated GUI can be coupled with a stepwise simulator to perform graphical simulations of models.

In fact, the main goal of such a DSL is to visualize the (dynamical) behavior of physical models. This visualization is particularly helpful during the design phases of complex systems. Until now, no common language dedicated to the graphical animation of models (independent of the application domain) has been designed. Our objective is to propose a DSL for graphical animation of models.

The remainder of this article is organized as follows. Section 2 gives an overview of the AltaRica 3.0 project. Section 3 presents the motivations of this work. Section 4 introduces GraphXica – a language for graphical animation of models. Section 5 summarizes the related works. Finally Section 6 concludes the article and outlines directions for future works.

2 ALTARICA 3.0 PROJECT

The objective of the AltaRica 3.0 project is to propose a set of authoring, simulation and assessment tools to perform Model-Based Safety Analyses. The overview of the project is presented Figure 1.

The new version of AltaRica modeling language, AltaRica 3.0, is in the heart of this project. It supports modeling of looped systems and bidirectional flows. This new version significantly increases the expressive power of the previous one without decreasing the efficiency of the assessment algorithms. AltaRica 3.0 models are compiled into a low level formalism: Guarded Transition Systems (Rauzy 2008, Prosvirnova and Rauzy 2012). Guarded Transition Systems is a states/transitions formalism that generalizes classical safety formalisms, such as Reliability Block Diagrams, Petri Nets and Markov chains. It is a pivot formalism for Safety Analyses: other safety models, not only AltaRica 3.0, can be compiled into Guarded Transition Systems to take benefits from the assessment tools. The assessment tools for Guarded Transition Systems already include prototypes of a compiler to Fault Trees, a compiler to Markov chains, a stochastic and a stepwise simulators. Prototypes of a model-checker and a reliability allocation module are planed to be developed. Distributed under a free license, the assessment tools enable users to perform virtual experiments on systems, via models, to compute different kinds of reliability indicators and, also, to perform cross check calculations.

3 MOTIVATIONS

As a part of AltaRica 3.0 Project our team works on the graphical simulation of AltaRica models. Graphical simulation of models has its own interest. First of all, it enables to better understand the system behavior. Second, virtual experiments can be performed on systems, via models, for example, it is possible to play calculated failure scenarii. Finally, it helps to debug and to validate the model.

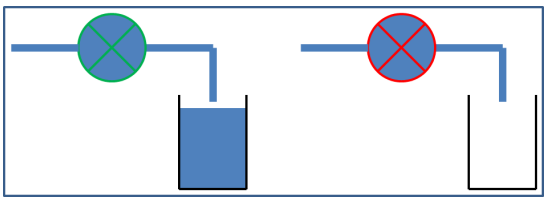


Figure 2: Graphical representation and animation of a water supply system

Consider a simple water supply system, composed of a pump and a tank. A pump can be in two states: WORKING or FAILED. If the pump is in state WORKING, then the tank is full, otherwise, it is empty. This system is represented in AltaRica as follows:

```
domain PumpState {WORKING, FAILED}
class Pump
  PumpState state (init = WORKING);
  Real input (reset = 0.0), output (reset = 0.0);
  event failure (delay = exponential(0.0005));
  event repair (delay = exponential(0.02));
  transition
    failure: state==WORKING -> state := FAILED;
    repair: state==FAILED -> state := WORKING;
  assertion
    output := if (state==WORKING) then input
              else 0.0;
end
class Tank
  Real input (reset = 0.0);
end
block WaterSupplySystem
  Pump pump;
  Tank tank;
  Real input (reset = 1.0);
  observer Boolean tank.isEmpty =
                          (tank.input == 0.0);
  assertion
    pump.input := input;
    tank.input := pump.output;
end
```

Our goal is to perform graphical simulation of this model. For that we need to define a graphical representation of the model and its animation, i.e. how the representation changes according to the values of system variables (see for example Figure 2). Basically, we would like to define the following animations:

1. If the pump is failed (*pump.state == FAILED*), then change the outline color to red.
2. If the tank is empty (*tank.input == 0.0*), then hide the blue rectangle.

To be able to perform graphical animations of models we will

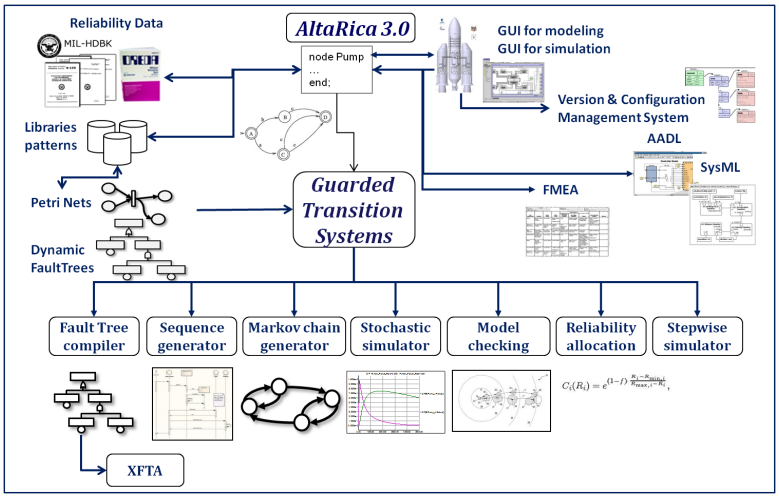


Figure 1: Overview of the AltaRica 3.0 project

- First, define a Domain Specific Language (DSL) to describe graphical representations and animations of models.
- Then, use this DSL to generate Graphical User Interfaces (GUIs) for event-driven simulation of models.
- Finally, couple the generated GUI with a stepwise simulator of AltaRica.

3.1 *DSL for graphical animation of models*

DSL for graphical animation should provide specific primitives to represent graphical objects (i.e. figures) and to describe their animations.

Graphical primitives Graphical objects give the static representation of the model (i.e. its 2D or 3D representation). The language should include at least the following graphical objects:

- Basic geometric figures: Rectangle, Ellipsoid, Line, etc.
- Links: Line, Point (to represent connections between graphical objects).
- Text (to display textual annotations of models).
- Bitmap (to use predefined pictures).

All graphical primitives should have some shared attributes, such as size, color, hidden, opacity, etc.

Composition It should be possible to design libraries of reusable graphical components (i.e. figures and their animations) and to assemble them in order to create graphical representations and animations of systems. In the example given Figure 2, one should be able to create graphical representations of a pump and a tank, their animations and to use them to create the graphical representation of the system, composed of the pump and the tank connected together.

Animation The ability to describe the animations of the model, i.e. how changes the graphical representation of the model in time, is the most important part of the language. Graphical animations of models should be done in two ways:

- Depending on external variables.
- According to user actions.

In the first case, the animation may depend on some external variables. When these variables change their values, the graphical representation is updated according to the defined animation rules. The values of these variables may be obtained from a model simulator, such as, for example, the stepwise simulator of AltaRica 3.0, from a file or a database, etc. The second way of animation is done through a Graphical User Interface (GUI) that gives the user the ability to interact with the model.

Animations are related to previously defined graphical primitives (figures). We should consider at least the following animations for them:

- Move.
- Hide or Show.
- Modify attribute values (e.g. color, size, etc.).

3.2 *Graphical user interfaces for simulation*

The generated GUI for simulation can be implemented in different programming languages, for example:

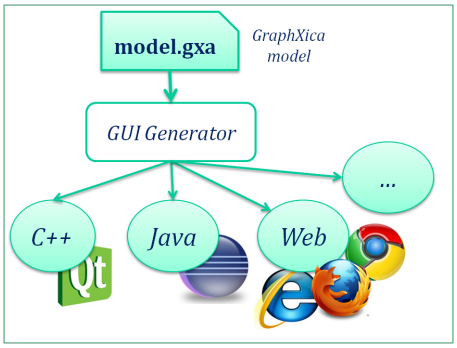


Figure 3: DSL for graphical animation of models

- All programming languages with Graphical libraries: C++ with Qt, C with SDL, Java with its standard graphics library, etc.
- Html and CSS to use with a web browser.
- Visual basic to use with the presentation program PowerPoint.

The GUI generator takes a graphical representation and animation of an AltaRica 3.0 model, generates a GUI that displays graphical animations of the model, as illustrated Figure 3.

3.3 *Graphical animation of AltaRica 3.0 models*

The stepwise simulator of AltaRica can be coupled with a graphical simulator in order to perform graphical simulation of models (e.g. see Perrot et al. (2010)). The link between the stepwise simulator and simulation GUI is done by a communication protocol.

Given an AltaRica model, it will be possible to define its graphical representation and animation using the DSL. Then a simulation GUI will be generated; it will be linked with a stepwise simulator by a communication protocol in order to perform graphical simulation of AltaRica models. The procedure is illustrated Figure 4.

4 GRAPHXICA

In this section we present GraphXica – a Domain Specific Language (DSL) for graphical animation of models. As shown figure 4, we use GraphXica with the stepwise simulator of the AltaRica 3.0. The generated simulation GUI, so called GraphXica Displayer, is implemented in Java. It takes a GraphXica model, corresponding to the graphical representation of the AltaRica 3.0 model. According to fired transitions, the stepwise simulator emits a vector of variables values. The GraphXica Displayer receives this vector and performs animations of figures. These animations are defined by the animation rules and depend on the values of variables received from the stepwise simulator.

GraphXica is, like AltaRica 3.0, a prototype oriented modeling language, see e.g. Noble et al. (1999) for a discussion on objects versus prototypes. Prototype orientation makes it possible to separate the knowledge into two distinct spaces: the stabilized knowledge, incorporated into libraries of on-the-shelf modeling components; the sandbox in which the system under study is modeled. In the sandbox, many components are unique and some others are instances of reusable components. With prototype-orientation, models can be reused in two ways: at component level by instantiating on-the-shelf components; at system level by cloning and modifying a model designed for a previous project. Classes represent the stabilized knowledge: they can be instantiated and extended like in object-oriented languages. Blocks model unique components, that cannot be instantiated. The system is always represented by a block.

A GraphXica model is made up of declarations. It is possible to declare global variables, variable domains (i.e. enumerated types) and components (i.e. classes or blocks) to describe figures and their animation rules. In the following, we present the grammar in extended BNF.

Model ::= (*Declaration*)* ;
Declaration ::=
  DomainDeclaration
| *ExternVariableDeclaration*
| *VariableDeclaration*
| *ClassDeclaration*
| *BlockDeclaration*
;

4.1 *Domains*

Domains are named sets of symbolic constants. They are defined in the following way:

DomainDeclaration ::=
  'domain' *Identifier* '{' *Identifier* (',' *Identifier*)* '}' ;

An identifier is a letter followed (not necessary) by a sequence of letters or numbers. The special character '_' is also included to define an identifier. The rule *Identifier* is the following:

Identifier ::=
  (*Letter* | '_') (*Letter* | *Number* | '_')* ;
Letter ::= 'a' | ... | 'z' | 'A' | ... | 'Z' ;
Number ::= '0' |... | '9' ;

Declared domains can be used as a type for variables anywhere in the model. In the example given Figure 2, domains expressing the state of the pump and the tank are declared as follows:

```
domain PumpState {WORKING, FAILED}
domain TankLevel {FULL, EMPTY}
```

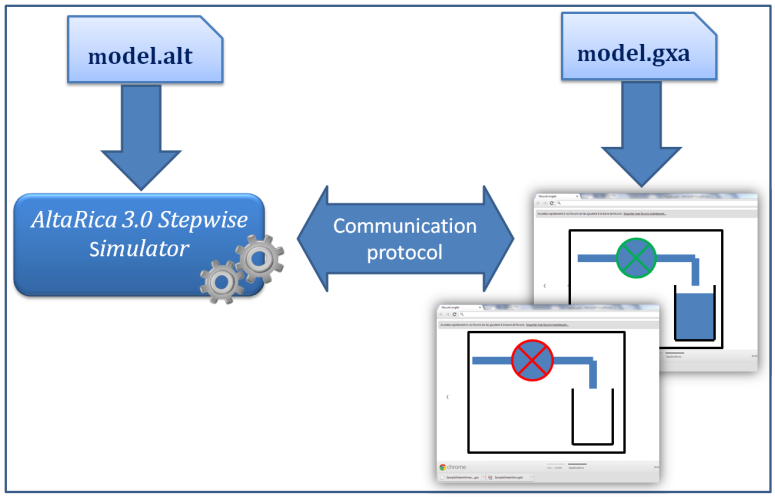


Figure 4: Graphical simulation of models

4.2 *Global variables*

Variables are declared in the following way:

ExternVariableDeclaration ::=
  'extern' *VariableDeclaration* ;
VariableDeclaration ::=
  Type Variable (',' *Variable*)* ';' ;
Type ::=
  NumericalType'
  | *FigureType*
  | *Identifier*
  ;
NumericalType ::=
  'Boolean'
  | 'Integer'
  | 'Real'
  ;
Variable ::=
  Identifier ['(' *Attributes* ')'] ;
Attributes ::=
  Attribute (',' *Attribute*)* ;
Attribute ::=
  Identifier '=' *Expression* ;

The rule *Expression* defines formulas built over variables and parameters using common arithmetic, comparison and logical operators. We will not detail it here. Global variables can only have numerical or user defined type. Thus, the rule *FigureType* will be defined later.

Global variables have the same sense like in the programming languages C, C++ or Java. A global variable can be used anywhere in the model. The keyword `extern` can be added to the declaration of a variable. It expresses the fact that the variable will be linked to a variable coming from the vector of values (from an assessment tool linked to the GraphXica Displayer). When a variable is declared, a set of attributes (e.g. its initial value) can also be declared. In the example given Figure 2, we will declare two external variables `ext_pumpState` and `ext_tankInput`, corresponding to variables from AltaRica 3.0 model:

```
extern PumpState ext_pumpState (init = WORKING);
extern Boolean ext_tankInput (reset = false);
```

4.3 *Classes and Blocks*

Classes are used to create libraries of reusable graphical representations and animations. Blocks are used to design graphical representations of systems using libraries of reusable components. They represent clearly the separation between stabilized knowledge, incorporated into libraries of on-the-shelf modeling components, and the sandbox in which the system under study is modeled. Classes are declared as follows:

ClassDeclaration ::=
  'class' *Identifier*
    (*ComponentDeclaration*)*
    [*Animations*]
  'end' ;

Blocks are declared as follows:

BlockDeclaration ::=
  'block' *Identifier*
    (*ComponentDeclaration*)*
    [*Animations*]
  'end' ;

Components are declared as follows:

ComponentDeclaration ::=
  VariableDeclaration
  | *ParameterDeclaration*
  ;

Classes may embed instances of other classes so to get hierarchical representations of systems under study. Blocks may be composed of other blocks and instantiated classes. Blocks and classes may contain variables, parameters, figures and animation rules. Here, types of variables can be figures type (e.g.: rectangle, oval, line, etc.) or user declared classes.

The following classes describe the graphical representations of the pump and the tank from the system given Figure 2.

```
class PumpRep
  parameter PumpState state  = WORKING;
  Oval cir (x = 0, y = 0, width = 5, height = 5,
       color = blue, lineColor = black,
       thickness = 2);
  Line delta1 (x = 1.5, y = 8.5, width = 7,
       height = -7, color = black, thickness = 2);
  Line delta2 (x = 1.5, y = 1.5, width = 7,
       height = 7, color = black, thickness = 2);
  animation
    state == FAILED ->
       {
         delta1.color := red;
         delta2.color := red;
         cir.lineColor := red;
       }
    state == WORKING ->
       {
         delta1.color := green;
         delta2.color := green;
         cir.lineColor := green;
       }
end

class TankRep
  parameter TankLevel level = FULL;
  Line horLeft (x = 0 ,y = 0, width = 14,
       height = 0, color = black,
       thickness = 2);
  Line horRight (x = 10 ,y = 0, width = 14,
       height = 0, color = black,
       thickness = 2);
  Line base (x = 0 ,y = 14, width = 0,
       height = 10, color = black,
       thickness = 2 );
  Rectangle rectFull (x = 0, y = 2,
       width = 10, height = 12,
       color = blue, thickness = 0,
       visible = true);
  animation
    level == EMPTY -> rectFull.visible := false;
    level == FULL -> rectFull.visible := true;
end
```

Graphical primitives GraphXica contains specific primitives to represent figures and to define their animations. Figures are declared as variables: they have a type, a name and a list of attributes. Different types of figures are included, such as, for example, `Line`, `Rectangle`, `Oval`, `Bitmap`, `Text`, etc. Each type of figure has its own list of attributes. However, there are some common attributes, such as the color of the figure, its position (with two dimensions), its size (with two dimensions), its visibility and its opacity. The idea is to consider a figure inscribed inside an "imaginary" rectangle (it is not a figure of the language) and positions are according to the left-top corner.

In the example given above a pump is represented by a circle (an oval) *cir* and two lines *delta1* and *delta2* and a tank is described by three lines *horLeft, horRight, base* and a rectangle *rectFull*. A line is defined by the coordinates of its source and its target, by its color and its thickness. A rectangle is defined by the coordinates of its left-top corner, its size, its color and the thickness of its border. Note, that users don't have to fill in all the attributes, GraphXica Displayer gives a default value for all unspecified attributes.

Parameters Parameters are introduced by a keyword **parameter**, followed by the type, the name and the value (i.e. an expression depending on variables and other parameters). They are declared in the following way:

ParameterDeclaration ::=
'parameter' *NumericalType Identifier* '=' *Expression* ';' ;

In the example given earlier we declare two parameters: *level* in the class *TankRep* and *state* in the class *PumpRep*. The values of this parameters can be changed when the classes are instantiated. Parameters can be used in different manners: they can be linked to external variables or they can define constant values, such as circle radius, rectangle height, etc.

Variables Local variables within a class or a block can be declared in the same way as global variables.

Animations Animations are defined by a set of rules. Each rule is represented by a condition followed by a set of instructions. Animations are declared in the following way:

Animations ::=
'animation' (*Animation*)* ;
Animation ::=
Condition '→' *Instruction* ;
Condition ::=
LogicalExpression ;
Instruction ::=
Assignment
| *Block*
;
Assignment ::= *Identifier* ':=' *Expression* ';' ;
Block ::= '{' *Instruction*+ '}' ;

Conditions are Boolean expressions, built over variables and parameters of the model (e.g. `state == FAILED`) or user actions (e.g. click on a figure). Boolean expressions are evaluated according to the vectors of values, received from a linked assessment

tool. Instructions enable to modify the values of attributes of figures (e.g. change the color or visibility of a figure, move or enlarge a figure, etc.). Of course, the modifications of attribute values are defined according to the considered figures. For example, it is possible to change the color of a **Rectangle**, but it is not possible to modify the color of a **Bitmap**.

In the example given above, the class *Pump* defines the following animation rules: if the value of the parameter *state* is WORKING, then the color of lines is green, otherwise it is red. In the class *Tank* if the value of the parameter *level* is EMPTY then the rectangle *rectFull* is hidden.

Composition To create a GraphXica model of the water supply system, given Figure 2, we use the previously defined classes *PumpRep* and *TankRep*. The block *WaterSupplySystemRep* is composed of an instance of class *PumpRep* and an instance of class *TankRep*. The parameter *state* of the *pumpRep* is set to the external variable `ext_pumpState` (this variable comes from AltaRica model of the system). The parameter *level* of the *tankRep* is calculated according to the external variable `ext_tankInput` coming from the AltaRica model.

```
block WaterSupplySystemRep
  TankRep tankRep (posX = 1, posY = 8,
      level = if ext_tankInput then FULL
                  else EMPTY);
  PumpRep pumpRep (x = 35, y = 2,
       state = ext_pumpState);
  Line hLineLeft (x = 0, y = 5, width = 10,
       height = 0, color = blue, thickness = 6 );
  Line vLineLeft (x = 30, y = 5, width = 0,
       height = 7, color = blue, thickness = 6 );
  Line hLineRight (x = 20, y = 5, width = 10,
       height = 0, color = blue, thickness = 6 );
end
```

The animation rules of the block are executed in the following way: first, animation rules of all instantiated classes and nested blocks are executed in the same order as they are declared in the block, then animation rules of the main block are performed.

5 RELATED WORKS

In this section we introduce some existing languages for graphical representation and animation of models.

Modelica (Fritzson 2004) is an object-oriented language based on equations for modeling continuous or discrete event behavior of physical systems. The grammar of Modelica, amongst others, defines annotations used to store additional information about the model, such as its graphical representation. They are used to graphically represent a model and its components by means of graphical objects (rectangles, circles, etc.), component icons and connection lines. Although Modelica's annotations define all the necessary properties and primitives to represent a model in a graphical way. These representations are purely static. There is no animation of models and also no interaction between the user and a model.

Some existing languages are specifically dedicated to animation of algorithms: JAWAA (Rodger 2002), XAAL (Karavirta 2005), AnimalScript (Rößling and Freisleben 2001), JSamba (Stasko 1998), JHAVÉ (Naps et al. 2000). They are scripting languages for creating animation of algorithms. They contain primitives to represent graphical objects such as circles, rectangles, lines, etc.; and to animate them. The main principle is to write, or automatically generate, a script from an algorithm. This script corresponds to a translation from the algorithm to its graphical representation and animation. The script is then used by a "displayer", e.g. a web browser for JAWAA. Nevertheless, no communication is possible between the "displayer" and the user or another tool. For example, if one want to change the value of a variable, he has to rewrite the script.

SVG, for Scalable Vector Graphics (Eisenberg 2002), is a language for describing two-dimensional graphics in XML. SVG allows to create graphical forms (e.g.: circles, polygon, etc.), images and texts. SVG representations (i.e. drawing) are interactive. They can react to user's actions such as pressed button by the mouse.

The study of existing languages containing a part dedicated to graphical animation and the limited domain of use of these languages give us the reason to create a new language of graphical animation of models (GraphXica).

6 CONCLUSION AND PERSPECTIVES

In this article, we introduced GraphXica – a high-level modeling language for graphical animation of models. GraphXica enables to describe graphical representations of models and their animations. Animations are defined according to values of external variables and user actions. GraphXica models can be used to generate different types of Graphical User Interfaces (e.g. a Java interface, a web based interface, etc.). The generated GUI, so called GraphXica displayer, can be coupled with simulators in order to perform graphical simulations of models. GraphXica is a prototyped based modeling language. Thus, it is possible to create libraries of reusable graphical components and to use them to design graphical representations and animations of complex systems.

This work is done as a part of AltaRica 3.0 Project which aims to propose a complete set of authoring, simulation and assessment tools to perform Model-Based Safety Analyses. The idea is to couple GraphXica displayer with AltaRica 3.0 stepwise simulator in order to perform graphical simulation of models.

GraphXica displayers, implemented in Java and in C++ Qt, are currently under development. We also

plan to implement a web based GraphXica displayer using HTML 5 and CSS. Forthcoming papers will completely present the grammar of GraphXica and its semantic. Furthermore, our future works will focus on the definition and on the implementation of the communication protocol between GraphXica displayers and stepwise simulator. The redaction of pedagogical materials for GraphXica, including a primer, a best practices guide and a book of exercises is also an important part of the project. These materials will help interested persons to quickly understand and learn how to use GraphXica. Another interesting tools will be a generator of GraphXica models from AltaRica 3.0 models and an authoring tool for GraphXica.

REFERENCES

Bernard, R., J.-J. Aubert, P. Bieber, C. Merlini, & S. Metge (2007). Experiments in model-based safety analysis: flight controls. In *Proceedings of IFAC workshop on Dependable Control of Discrete Systems, Cachan*.

Bieber, P., J.-P. Blanquart, G. Durrieu, D. Lesens, J. Lucotte, F. Tardy, M. Turin, C. Seguin, & E. Conquet (2008, January). Integration of formal fault analysis in assert: Case studies and lessons learnt. In *Proceedings of 4th European Congress Embedded Real Time Software, ERTS 2008*, Toulouse (France).

Boiteau, M., Y. Dutuit, A. Rauzy, & J.-P. Signoret (2006). The altarica data-flow language in use: Assessment of production availability of a multistates system. *Reliability Engineering and System Safety 91*, 747–755.

Eisenberg, J. (2002, February). *SVG Essentials*. O'Reilly Media.

Fritzson, P. (2004). *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. John Wiley & Sons Inc.

Karavirta, V. (2005). Xaal - extensible algorithm animation language. Master's thesis, Helsinki University of Technology.

Naps, T., J. Eagan, & L. Norton (2000). JHAVÉ: An environment to actively engage students in web-based algorithm visualizations. *31st ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, 109–113.

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

Perrot, B., T. Prosvirnova, A. Rauzy, J.-P. S. d'Izarn, & R. Schoening (2010, October). Expériences de couplages de modèles AltaRica avec des interfaces métiers. In E. Fadier (Ed.), *Actes du congrès LambdaMu'17 (actes électroniques)*. IMdR.

Prosvirnova, T. & A. Rauzy (2012, Octobre). Guarded transition systems: Pivot modelling formalism for safety analysis. In J. Barbet (Ed.), *Actes du Congrès LambdaMu 18*.

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

Rauzy, A. (2008). Guarded transition systems: a new states/events formalism for reliability studies. *Journal of Risk and Reliability 222*(4), 495–505.

Rodger, S. (2002, June). Using hands-on visualizations to teach computer science from beginning curses to advanced courses. In *Proceeding of the Second Program Visualization Workshop*.

Rößling, G. & B. Freisleben (2001). Animalscript: An extensible scripting language for algorithm animation.

Stasko, J. (1998). Smooth continuous animation for portraying algorithms and processes. In *Software Visualization*, pp. 103–118. MIT Press.

Appendix D

Modeling patterns

One cannot expect models of complex systems to be simple. To capture (at least some aspects of) the complexity of the system, they need to be large and complex. The process by which they are designed is thus necessarily complex as well. Therefore, not only suitable modeling languages and efficient assessment tools must be used, but well-defined modeling methodologies must be applied so as to make this process effective.

In this article, we advocate that it is nearly impossible to get the good model for a system at once for at least two reasons: first, calculations of reliability indicators are provably hard (#P-hard for most of them [104]). Therefore, a model is always a tradeoff between the accuracy of the description and the ability one has to perform calculations within reasonable amounts of time and resources. Obviously, one cannot expect to reach the good tradeoff without some (and often more than many) trials.

Second, the different assessment tools at hand have their own advantages and drawbacks. So it is worth to use them in turn. However assessment tools put constraints on the model: compilation into Fault Trees makes it possible to handle very large models, but prevent from taking into account dependencies amongst failures of components; at the other extreme, stochastic simulation is a very versatile tool, but cannot be applied to capture low probabilities.

In a word, a Reliability and Safety Assessment should better consist in the design of a family of models rather than in the design of a unique model. This fact should be accepted by the analysts and be at the core of the modeling methodologies.

Nevetheless, this raises in turn the question of how to relate and to maintain these models throughout the life cycle of the systems. Thanks to our red wire example, we shall illustrate that AltaRica 3.0 is of a great help for that purpose. We show how, starting from the same root model, different variants can be obtained by successive refinements. Each variant, or subset of variants, is tailored for a particular assessment tool, i.e. to capture a particular aspect of the system under study.

Safety Assessment of an Electrical System with AltaRica 3.0

Hala Mortada[1], Tatiana Prosvirnova[1], and Antoine Rauzy[2]

[1] Computer Science Lab, Ecole Polytechnique, Route de Saclay, Palaiseau, France
Hala.Mortada@lix.polytechnique.fr Prosvirnova@lix.polytechnique.fr
[2] Chaire Blériot Fabre, LGI Ecole Centrale de Paris Grande voie des vignes, 92295 Châtenay-Malabry, France Antoine.Rauzy@ecp.fr

Abstract. This article presents the high level, modeling language AltaRica 3.0 through the safety assessment of an electrical system. It shows how, starting from a purely structural model, several variants can be derived.Two of them target a compilation into Fault Trees and two others target a compilation into Markov chains. Experimental results are reported to show that each of these variants has its own interest. It also advocates that this approach made of successive derivation of variants is a solid ground to build a modeling methodology onto.

Keywords: AltaRica3.0, Complex systems, Reliability, Modeling, Safety

1 Introduction

The increasing complexity of industrial systems calls for the development of sophisticated engineering tools. This is indeed true for all engineering disciplines, but especially for safety and reliability engineering. Experience shows that traditional modeling formalisms such as Fault Trees, Petri nets or Markov processes do not allow a smooth integration of risk analysis within the overall development process. These analysis require both considerable time and expertise. The specialization and the lack of model's structures make it difficult to share models amongst stakeholders, to maintain them throughout the life-cycle of the systems, and to reuse them from one a project to another.

The AltaRica modeling language ([1],[2]) has been created at the end of the nineties to tackle these problems. AltaRica makes it possible to design high-level models with a structure that is very close to the functional or the physical architecture of the system under study. Its constructions allow models to be structured into a hierarchy of reusable components. It is also possible to associate graphical representations to these components in order to make models visually close to Process and Instrumentation Diagrams. The formal semantics of AltaRica allowed the development of a versatile set of processing tools such as compilers into Fault Trees ([2]), model-checkers ([3]) or stochastic simulators ([4]). A large number of successful industrial experiments with the language have been reported (see e.g. [5], [6], [7], [8] and [9]). Despite its quality, AltaRica faced two issues of very different natures. First, systems with instant feedback's loops

turned out to be hard to handle. Second, constructs of model's structuring were not fully satisfying.

AltaRica 3.0 [10] is a new version of the language that has been designed to tackle these two issues. Regarding model structuring, AltaRica 3.0 implements the prototype-oriented paradigm [11]. This paradigm fits well with the level of abstraction reliability and safety analysis stand at. Regarding mathematical foundations, AltaRica 3.0 is based on Guarded Transition Systems (GTS) [12]. GTS combine the advantages of state/event formalisms such as Petri nets and combinatorial formalisms such as block diagrams. This combination is necessary to model system patterns namely cold redundancies, cascading failures or remote interactions.

AltaRica 3.0 comes with a variety of assessment tools. In this article, we show how, starting from the same root model, different variants can be obtained by successive refinements: a first series targeting a compilation into Fault Trees and a second one targeting a compilation into Markov chains. Each of these variants capture a particular aspect of the system under study. We advocate that this approach made of successive derivation of variants is a solid ground to build a modeling methodology onto.

The remainder of this article is organized as follows. Section 2 presents the electrical system that is used as a red-wire example throughout the paper. Section 3 discusses how to describe the architecture of the system with a purely structural model. Section 4 proposes a first variant of this structural model which targets a compilation into Fault Trees. Section 5 presents a second variant which targets a compilation into Markov Chains. Finally, section 6 concludes this article.

2 Red Wire Example

Figure 1 shows a simple electrical system with cascade redundancies borrowed from [13] (we present it here with some additional complexity).

In a normal operating mode, the busbar BB is powered by the grid GR either through line 1 or through line 2. Each line is made of an upper circuit breaker CBUi, a transformer TRi and a lower circuit breaker CBDi. The two lines are in cold redundancy: Let's assume for instance that line 1 was working and that it failed either because one of the circuit breakers CBU1 or CBD1 failed, or because the transformer failed. In this case, the line 2 is attempted to start. This requires opening the circuit breaker CBD1 (if possible/necessary) and closing the circuit breakers of line 2. Since line 2 was out of service, the circuit breaker CBD2 was necessarily open.

If both lines fail, the diesel generator DG is expected to function, which requires closing the circuit breaker CB3. Circuit breakers may fail to open and to close on demand. The diesel generator may fail on demand as well. The grid GR may be lost either because of an internal failure or because of a short circuit in the transformer TRi followed by a failure to open the corresponding circuit breaker CBUi.

The two transformers are subject to a common cause failure.

There is a limited repair crew that can work on only two components at a time. After maintenance, the components are as good as new, but may be badly reconfigured.

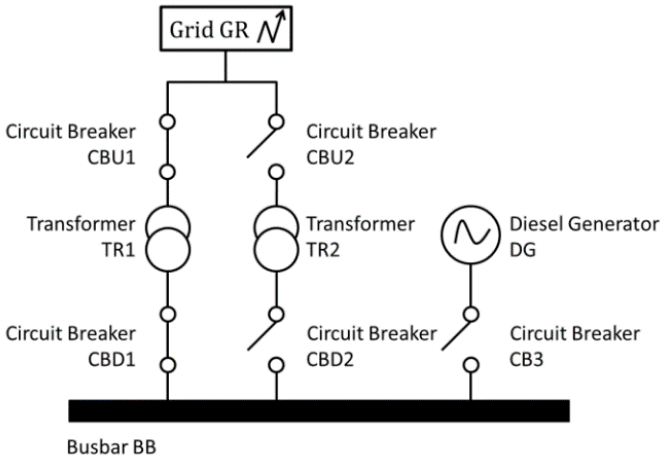


Fig. 1. A small electrical system

The problem is to estimate the reliability and the availability of this system. This example is small but concentrates on a number of modeling difficulties (warm redundancies, on demand failures, short-circuit propagation, common cause failures, limited resources), due to its multi-domains aspects.

3 Describing the Architecture of the System

The first step in analyzing a system consists of describing its functional and physical architecture. Figure 1 describes a possible decomposition of our electrical system. This decomposition deserves three important remarks.

First, it mixes functional and physical aspects. In fact, due to the small size of the example, only basic blocks (leaves of the decomposition) represent physical components. The others represent functions. We could consider functional and physical architectures separately. However, considering both in the same diagram simplifies things here. Moreover, it matches better with the usual way of designing models for safety and dependability analysis. Note also that at this step, we do not consider interactions between components.

Second, the underlying structure of this decomposition is not a tree, but a directed acyclic graph for the external power supply is shared between Line 1 and Line 2. As we shall see, this has very important consequences in terms of structuring constructs.

Third, the system embeds five circuit breakers and two transformers. We can assume that the circuit breakers on the one hand, the transformers on the other hand are all the same. From a modeling point of view, it means that we need to be able to define generic components and to instantiate them in different places in our model. On the contrary, components like "Primary Power Supply", "Backup Power Supply" and "Busbar Power Supply" are specific to that particular system.

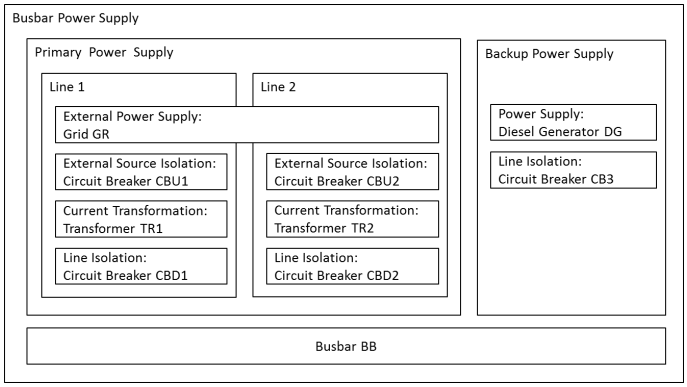


Fig. 2. Architecture of the Busbar Power Supply System

The structure of the AltaRica 3.0 model that reflects this architecture is sketched in Figure 2. In AltaRica 3.0, components are represented by means of blocks. Blocks contain variables, events, transitions, and everything necessary to describe their behavior. At this step, the behavioral part is still empty. Blocks can also contain other blocks and form hierarchies. The block "BusbarPowerSupply" contains two blocks: "PrimaryPowerSupply" and "BackupPowerSupply". "BusbarPowerSupply" is the parent block of "PrimaryPowerSupply" and an ancestor of "CBU1". Objects defined in a block are visible in all its ancestors. For instance, if the class "CircuitBreaker" defines an event "failToOpen", the instantiation of this event in "CBU1" is visible in the block "BusbarPowerSupply" through the dot notation, i.e. "PrimaryPowerSupply.Line1.CBU1.failToOpen".

An instance "GR" of the class "Grid" is declared in the block "PrimaryPowerSupply". It is convenient to be able to refer to it as "GR" as if it was declared in "Line1". This is the purpose of the "embeds" clause. This clause makes it clear that "GR" is part of "Line1", even if it is probably shared with some sibling blocks.

Classes in AltaRica 3.0 are similar to classes in object-oriented programming languages (see e.g. [14], [15] for conceptual presentations of the object-oriented paradigm). A class is a block that can be instantiated, i.e. copied, elsewhere in the model. There are several differences however between blocks and classes. AltaRica 3.0 makes a clear distinction between "on-the-shelf", stabilized knowl-

edge, for which classes are used, from the model itself, i.e. the implicit main block and all its descendants. Such a distinction has been conceptualized in CK-Theory ([16]). The implicit main block can be seen as the sandbox in which the analyst is designing his model. Declaring a class is in some sense creating another sandbox. Amongst other consequences, this means that it is neither possible to refer in a class to an object which is declared outside of the class, nor to declare a class inside another one or in a block. A class may of course contain blocks and instances of other classes up to the condition that this introduces no circular definition (recursive data types are not allowed in AltaRica 3.0). To summarize, AltaRica 3.0 borrows concepts to both object-oriented programming and prototype-oriented programming [11] - blocks can be seen as prototypes - so to provide the analyst with powerful structuring constructs that are well suited for the level of abstraction of safety analysis.

```
block BusbarPowerSupply
  block PrimaryPowerSupply
    Grid GR;
    block Line1
      embeds GR;
      CircuitBreaker CBU1, CBD1;
      Transformer TR1;
    end
    block Line2
      embeds GR;
      CircuitBreaker CBU2, CBD2;
      Transformer TR2;
    end
  end
  block BackupPowerSupply
    DieselGenerator DG;
    CircuitBreaker CB3;
  end
end
class Grid
end
...
```

Fig. 3. Structure of the AltaRica 3.0 Model for the Electrical System (partial view)

4 Targeting Compilation into Fault Trees

4.1 A Simple Block-Diagram like Model

We shall consider first a very simple model, close to a block diagram, in which basic blocks have a (Boolean) input, a (Boolean) output and an internal state

(WORKING or FAILED). This basic block changes its state, from WORKING to FAILED, when the event "failure" occurs. The diagram for the whole system is pictured in Figure 4.

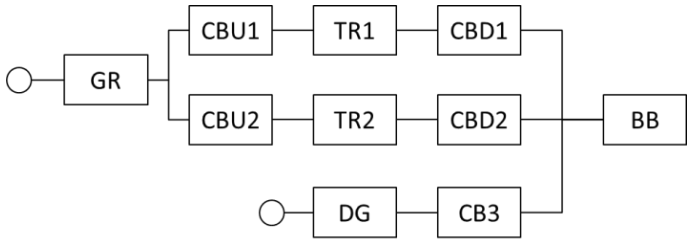


Fig. 4. A Block Diagram for the electric supply system

The AltaRica code for the diagram (with the architecture defined in the previous section) is sketched in Figure 5. The class "NonRepairableComponent" declares a state variable "s" and two Boolean flow variables: "inFlow" and "outFlow". "s" takes its value in the domain "ComponentState" and is initially set to WORKING. "inFlow" and "outFlow" are (at least conceptually) reset to false after each transition firing. Their default value is false. The class also declares the event "failure" which is associated with an exponential probability distribution of parameter "lambda". This parameter has the value "1.0e-4" unless stated otherwise.

After the declaration part, which consists in declaring flow and state variables, events and parameters, comes the behavioral part. This behavioral part itself includes transitions and assertions. In our example, there is only one transition and one assertion. The transition is labeled with the event "failure" and can be read as follows. The event "failure" can occur when the condition "s == WORKING" is satisfied. The firing of the transition gives the value FAILED to the variable "s".

The assertion describes the action to be performed to stabilize the system after each transition firing (and in the initial state). In our example, the variable "outFlow" takes the value true if "s" is equal to WORKING and "inFlow" is true, and false otherwise. In the initial state, all components are working, so the value true propagates from the input flow of the grid "GR" to the output flow of the system. If the circuit breaker "CBU2" fails, then the value false propagates from the output flow of "CBU2" to the output flow of the Line 2.

It would be possible to copy-paste the declaration of "NonRepairableComponent" in the declaration of the basic components of our model ("Grid", "CircuitBreaker", etc.). However, AltaRica 3.0 is an object-oriented language and thus provides a much more elegant way to obtain the same result: inheritance. It suffices to declare that the class "Grid" inherits from class "NonRepairableComponent". This is done in the code of Figure 5. In the class "Grid" the default value of the input flow is set to "true". "This change makes the grid a source

```
domain ComponentState { WORKING, FAILED }
class NonRepairableComponent
  Boolean s (init = WORKING);
  Boolean inFlow, outFlow (reset = false);
  event failure (delay = exponential(lambda));
  parameter Real lambda = 0.0001;
  transition
    failure: s == WORKING -> s := FAILED;
  assertion
    outFlow := s == WORKING and inFlow;
end
class Grid extends NonRepairableComponent(inFlow.reset = true);
end
...
block BusbarPowerSupply
  Boolean outFlow(reset = false);
  Grid GR;
  block PrimaryPowerSupply
    Boolean outFlow (reset = false);
    block Line1
      Boolean outFlow (reset = false);
      embeds GR;
      CircuitBreaker CBU1, CBD1;
      Transformer TR1;
      assertion
        CBU1.inFlow := GR.outFlow;
        ...
    end
    block Line2
      ... // similar to Line1
    end
    assertion
      outFlow := Line1.outFlow or Line2.outFlow;
  end
  ...
  assertion
    outFlow := PrimaryPowerSupply.outFlow or BackupPowerSupply.outFlow;
end
```

Fig. 5. A simple model targeting a compilation into Fault Trees (partial view)

block. The remainder of the model consists in plugging inputs and outputs of the components in order to build the system. Note that the resulting model is not just a flat block diagram, but a hierarchical one. The compilation of this model into Fault Trees is performed according to the principle defined in [2]. The idea is to build a Fault Tree such that:

– The basic events of this Fault Tree are the events of the AltaRica model.

– There is (at least) an intermediate event for each pair (variable, value) of the AltaRica model.
– For each minimal cutset of the Fault Tree rooted by an intermediate event (variable, value), there exists at least one sequence of transitions in the AltaRica model labeled with events of the cutset that ends up in a state where this variable takes this value. Moreover, this sequence is minimal in the sense that no strict subset of the minimal cutsets can label a sequence of transitions ending up in a state where this variable takes this value.

For technical reasons, the Fault Trees generated by the AltaRica compiler are quite different from those an analyst would write by hand. The minimal cutsets are however the expected ones. For instance, the minimal cutsets for the target "(BusbarPowerSupply.outFlow, false)", i.e. the busbar is not powered, with our first model are as follows.

GR.failure DG.failure	GR.failure CB3.failure
CBU1.failure CBU2.failure DG.failure	CBU1.failure TR2.failure DG.failure
CBU1.failure CBU2.failure CB3.failure	TR1.failure CBD2.failure DG.failure
CBU1.failure CBD2.failure CB3.failure	TR1.failure CBU2.failure DG.failure
CBU1.failure CBD2.failure DG.failure	TR1.failure CBU2.failure CB3.failure
CBD1.failure CBU2.failure DG.failure	TR1.failure TR2.failure CB3.failure
CBU1.failure TR2.failure CB3.failure	TR1.failure CBD2.failure CB3.failure
CBD1.failure CBU2.failure CB3.failure	TR1.failure TR2.failure DG.failure
CBD1.failure CBD2.failure DG.failure	CBD1.failure TR2.failure CB3.failure
CBD1.failure CBD2.failure CB3.failure	CBD1.failure TR2.failure DG.failure

4.2 Taking into account Common Cause Failures

We shall now design a second model in order to take into account the common cause failure of the two transformers (due for instance to fire propagation). To do so, we have to model that transformers fail simultaneously when the common cause failure occurs. AltaRica provides powerful synchronization mechanisms to make transitions simultaneous. The idea is to create an event "CCF" and a transition at the first common ancestor of the two transformers, i.e. "PrimaryPowerSupply". The new code for the "PrimaryPowerSupply" is sketched in Figure 6. The operator & synchronizes the transitions "failure" defined for each transformer. The operator & is associative and commutative. Any number of transitions can be thus synchronized. To fire the synchronizing transition, at least one of the synchronized transitions must be fireable. If the synchronizing transition is fired, then all the possible synchronized transitions are fired simultaneously. The modality ? indicates that the corresponding synchronized transition is not mandatory to fire the synchronizing transition. The modality ! indicates that the corresponding transition is mandatory.

Note that the synchronized transitions continue to exist independently of the synchronizing transition. It is possible to hide transitions by means of a special clause "hide". Our second model has the following two additional minimal cutsets.

```
block PrimaryPowerSupply
  ...
  event CCF (delay = exponential(lambdaCCF));
  parameter Real lambdaCCF = 1.0e-5;
  ...
  transition
    CCF: ?Line1.TR1.failure & ?Line2.TR2.failure;
  assertion
  ...
end
```

Fig. 6. Synchronization mechanism to model the Common Cause Failures

PrimaryPowerSupply.CCF, BackupPowerSupply.DG.failure
PrimaryPowerSupply.CCF, BackupPowerSupply.CB3.failure

5 Targeting Compilation into Markov Chains

In this section we consider repairs of components, reconfigurations and limited resources. First, we assume that there is an unlimited number of repairers. Then, we refine our model to take into account a limited number of repairers. Both models are compiled into Markov chains.

5.1 Unlimited number of repairers

All components are repairable. The AltaRica code in this case is similar to the one of the "NonRepairableComponent" (see Figure 5), except that a new event "repair", the corresponding parameter mu and the corresponding transition are added to the previous model. Instead of the "NonRepairableComponent", the classes "Transformer" and "Grid" of this model, will inherit from a "Repairable-Component".

The on demand failures of the circuit breakers and the diesel generator are also considered. The automata describing the behavior of the diesel generator, the transformer, the grid and the circuit breakers are figured in 7 and 8. The solid lines correspond to the stochastic transitions, whereas the dashes correspond to the immediate ones.

Figure 9 represents the AltaRica 3.0 model of the spare component corresponding to the left automaton depicted in Figure 7. Transitions "stop", "start" and "failureOnDemand" are immediate (their delays are equal to 0). When the state variable "s" is equal to STANDBY and the flow variable "demanded" is true, the event "start" may occur with the probability "1-gamma" and the event "failureOnDemand" may occur with the probability "gamma". The values of the probabilities are given through the attribute "expectation".

In this example, we also take into consideration the short circuit case (see the automaton in the right hand side of the Figure 8). For the transformer, the

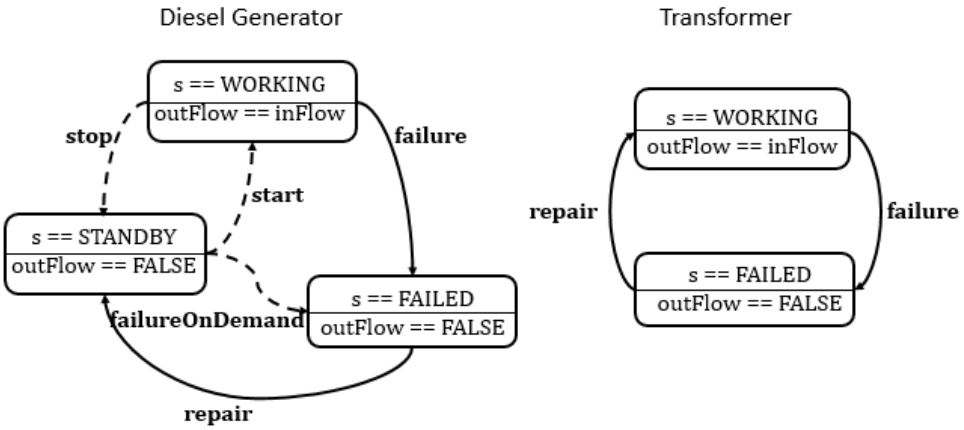


Fig. 7. Two automata describing the behavior of the Diesel Generator and the Transformer

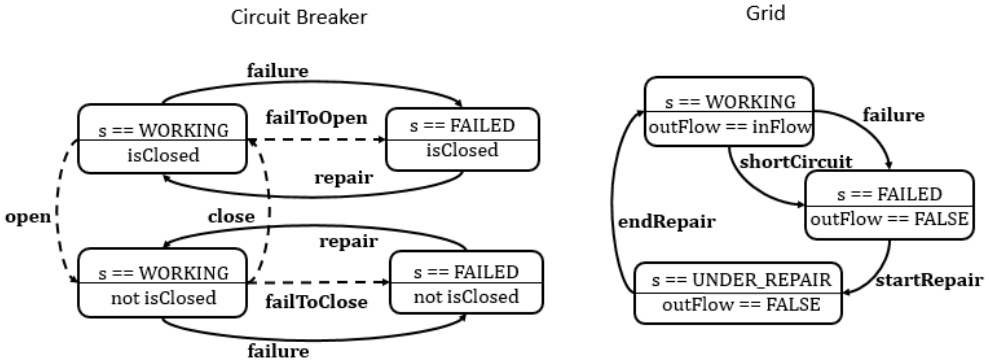


Fig. 8. Two automata describing the behavior of the Circuit Breaker and the Grid

event failure is considered as a short circuit, that will propagate into the whole line and make it instantly fail. If the short circuit is in the "Grid", the whole "Primary Power Supply" system will eventually fail, inducing the spare block (the "Backup Power Supply" system) to take over. The structure of the whole model remains the same as in Figure 3. Some additional assertions are added in order to represent the propagation of the short circuit from the transformers to the grid and the reconfigurations (orders to open/close circuit breakers, to start/stop the diesel generator).

The semantics of AltaRica 3.0 are a Kripke structure (a reachability graph) with nodes defined by variable assignments (i.e. variables and their values) and edges defined by transitions and labeled by events. If the delays associated to the events are exponentially distributed, then the reachability graph can be interpreted as a continuous time Markov chain. In the case when the graph

```
domain SpareComponentState { STANDBY, WORKING, FAILED }
class SpareComponent
  Boolean s (init = WORKING);
  Boolean demanded, inFlow, outFlow (reset = false);
  event failure (delay = exponential(lambda));
  event repair (delay = exponential(mu));
  event start (delay = 0, expectation = 1 - gamma);
  event failureOnDemand (delay = 0, expectation = gamma);
  event stop(delay = 0);
  parameter Real lambda = 0.0001;
  parameter Real mu = 0.1;
  parameter Real gamma = 0.001;
  transition
    failure: s == WORKING -> s := FAILED;
    repair: s == FAILED -> s := STANDBY;
    start: s == STANDBY and demanded -> s := WORKING;
    failureOnDemand: s == STANDBY and demanded -> s := FAILED;
    stop: s == WORKING and not demanded -> s := STANDBY;
  assertion
    outFlow := s == WORKING and inFlow;
end
```

Fig. 9. AltaRica 3.0 model of a spare component (Diesel generator)

contains immediate transitions, they are just collapsed using the fact that an exponential delay with rate λ followed by an immediate transition of probability p is equivalent to a transition with an exponential delay of rate $p\lambda$.

The generated Markov Chain contains 7270 states and 24679 transitions. The tool XMRK calculates the unavailability for different mission times. For $\lambda = 10^{-4}$, $\gamma = 10^{-3}$ and $\mu = 10^{-1}$, the probabilities are represented in Figure 12.

5.2 Limited number of repairers

In this part, we consider the case of a limited number of repairers, namely lower than the number of failures. Counter to the previous model, in order for a repair to take place, the repairer should be available and not used by another component. In this case, some changes in the behavior of the system take place. We will not only be interested in the "repair" transition, but also in the time it starts and ends at. Therefore, the "repair" transition is replaced by a whole set of transitions: startRepair and endRepair (see for example the automaton in the right hand side of the Figure 8). Besides, a new class called "RepairCrew" that defines when a job can start is added to the previous model (see Figure 10).

The transitions "startRepair" and "startJob", as well as "endRepair" and "endJob" are synchronized using the operator & as shown in Figure 11.

Compared to the definition of the common cause failure (see Figure 6), here the modality ! is used in the synchronization, which means that both synchro-

```
class RepairCrew
  Integer numberOfBusyRep (init = 0);
  parameter Integer totalNumberOfRepairers = 1;
  event startJob, endJob;
transition
  startJob: numberOfBusyRep < totalNumberOfRep ->
       numberOfBusyRep := numberOfBusyRep + 1;
  endJob: numberOfBusyRep > 0 ->
       numberOfBusyRep := numberOfBusyRep - 1;
end
```

Fig. 10. AltaRica model of the Repair Crew

```
block BusbarPowerSupply
  RepairCrew R;
  block PrimaryPowerSupply
    ...
  end
  block BackupPowerSupplySystem
    ...
  end
  event PPS_GR_startRepair, PPS_GR_endRepair;
  ...
transition
  PPS_GR_startRepair: !R.startJob & !PrimaryPowerSupply.GR.startRepair;
  PPS_GR_endRepair: !R.endJob & !PrimaryPowerSupply.GR.endRepair;
  hide R.startJob, PrimaryPowerSupply.GR.startRepair;
  hide R.endJob, PrimaryPowerSupply.GR.endRepair;
  ...
end
```

Fig. 11. A model targeting a compilation into Markov Chains (partial view)

nized events should be fireable to be able to fire the synchronizing transitions. In this example, the synchronized events are hidden explicitly using the clause "hide".

In order to make the results more interesting, two numbers of repairers $n = 1$ and $n = 3$ are considered. This will allow us to compare the two graphs of unavailability. The same parameters mentioned in the first subsection are used here as well. The Markov Chain consists of 29332 states and 98010 transitions. The graph in figure12 shows indeed that the unavailability is lower when the number of repairers is bigger, and even lower when it is unlimited.

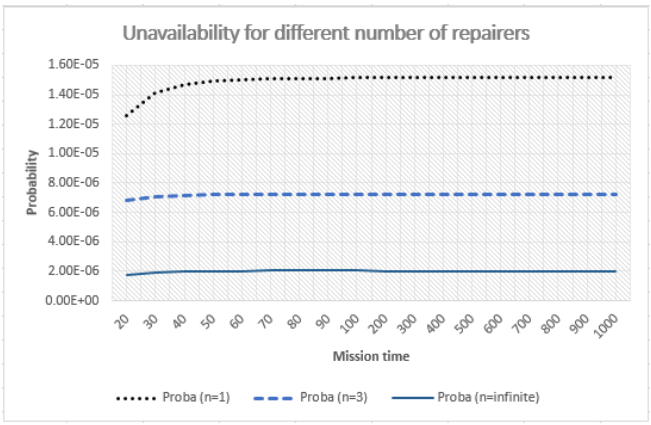


Fig. 12. Unavailability for a different number of repairers

6 Conclusion

In this paper we showed, using an electrical system as a red-wire example, how AltaRica 3.0 can be used to model complex phenomena. A purely structural model was designed. Then, we derived four variants from it: two of them targeting a compilation into Fault Trees and two others targeting a compilation into Markov chains. Each variant, or subset of variants, was tailored for a particular assessment tool, i.e. to capture a particular aspect of the system under study. Based on this experience (and several others we have performed), we are convinced that this approach, consisting of deriving models by means of successive refinements, is a solid ground to build a modeling methodology. The calculations to be performed are actually very resource consuming. Therefore, a model is always a trade-off between the accuracy of the description and the ability to perform calculations. Refining a model in successive variants is a good way to seek a good trade-off. Moreover, the trade-off depends on the characteristics of the system to be observed. Therefore, different tools must be applied. As a consequence, the refinement process should not be linear, but rather have a tree-like structure.

References

1. Arnold, A., Griffault, A., Point, G., Rauzy, A.: The altarica formalism for describing concurrent systems. Fundamenta Informaticae **34** (2000) 109–124
2. Rauzy, A.: Modes automata and their compilation into fault trees. Reliability Engneering and System Safety (2002)
3. Griffault, A., Vincent, A.: The mec 5 model-checker. In: Proceedings of the 16th International Conference on Computed Aided Verification (CAV 2004). Volume 3114., Boston MA, USA (2004) 488–491
4. Khuu, M.: Contribution à l'accélération de la simulation stochastique sur des modèles AltaRica Data Flow. PhD thesis, Université de la Mèditerranée, Aix-Marseille II (2008)

5. Humbert, S., Seguin, C., Castel, C., Bosc, J.M.: Deriving safety software requirements from an altarica system model. In: Proceedings SAFECOMP2008. Volume 5219., Newcastle upon Tyne, England (2008) 320–331
6. Quayzin, X., Arbaretier, E.: Performance modeling of a surveillance mission. In: Proceedings of the Annual Reliability and Maintainability Symposium, RAMS 2009, Fort Worth, Texas USA (2009) 206–211 ISBN 978-1-4244-2508-2.
7. Sghairi, M., De-Bonneval, A., Crouzet, Y., Aubert, J.J., Brot, P., Laarouchi, Y.: Distributed and reconfigurable architecture for flight control system. In: Proceedings of 28th Digital Avionics Systems Conference (DASC'09), Orlando, USA (2009)
8. Chaudemar, J.C., Bensana, E., Castel, C., Seguin, C.: Altarica and event-b models for operational safety analysis: Unmanned aerial vehicle case study. In: Proceedings Formal Methods and Tools, FMT'09, London, England (2009)
9. Adeline, R., Cardoso, J., Darfeuil, P., Humbert, S., Seguin, C.: Toward a methodology for the altarica modelling of multi-physical systems. In: Proceedings of European Safety and Reliability Conference, ESREL 2010, Rhodes, Greece (2010)
10. Prosvirnova, T., Batteux, M., Brameret, P.A., Cherfi, A., Friedlhuber, T., Roussel, J.M., Rauzy, A.: The altarica 3.0 project for model-based safety assessment. In: Proceedings of 4th IFAC Workshop on Dependable Control of Discrete Systems, DCDS'2013, York, Great Britain, International Federation of Automatic Control (2013) 127–132 ISBN: 978-3-902823-49-6, ISSN: 1474-6670.
11. Noble, J., Taivalsaari, A., Moore, I.: Prototype-Based Programming: Concepts, Languages and Applications. Springer-Verlag, Berlin and Heidelberg, Germany (1999) ISBN-10: 9814021253. ISBN-13: 978-9814021258.
12. Rauzy, A.: Guarded transition systems: a new states/events formalism for reliability studies. Journal of Risk and Reliability **222** (2008) 495–505
13. Bouissou, M., Bon, J.L.: A new formalism that combines advantages of fault-trees and markov models: Boolean logic-driven markov processes. Reliability Engineering and System Safety **82** (2003) 149–163
14. Meyer, B.: Object-Oriented Software Construction. Prentice Hall (1988) ISBN-10: 0136290493. ISBN-13: 978-0136290490.
15. Abadi, M., Cardelli, L.: A Theory of Objects. Monographs in Computer Science. Springer-Verlag. New York Inc (1998) ISBN-10: 0387947752. ISBN-13: 978-0387947754.
16. Hatchuel, A., Weil, B.: C-k design theory: an advanced formulation. research in engineering design. Research in Engineering Design **19** (2009) 181–192

Bibliography

[1] http://www.lix.polytechnique.fr/ rauzy/xfta/xfta.htm.

[2] Dubi A. *Monte Carlo application in Systems Engineering.* John Wiley and Sons Ltd., 2000.

[3] M. Abadi and L. Cardelli. *A theory of Objects.* Springer New York, 1996.

[4] Romain Adeline, Janette Cardoso, Pierre Darfeuil, Sophie Humbert, and Christel Seguin. Toward a methodology for the AltaRica modelling of multi-physical systems. In Ben J.M. Ale, Ioannis A. Papazoglou, and Enrico Zio, editors, *Proceedings of European Safety and Reliability Conference, ESREL 2010*, Rhodes, Greece, September 2010. Taylor and Francis Group. ISBN 978-0-415-60427-7.

[5] O. Akerlund, P. Bieber, E. Boede, M. Bozzano, M. Bretschneider, C. Castel, A. Cavallo, M. Cifaldi, J. Gauthier, A. Griffault, O. Lisagor, A. Luedtke, S. Metge, C. Papadopoulos, T. Peikenkamp, L. Sagaspe, C. Seguin, H. Trivedi, and L. Valacca. ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects. In *Proceedings of 3rd European Congress Embedded Real Time Software, ERTS 2006*, 2006.

[6] J.D. Andrews and T.R. Moss. *Reliability and Risk Assessment.* John Wiley & Sons, 1993. ISBN 0-582-09615-4.

[7] Andr Arnold, Alain Griffault, Gérald Point, and Antoine Rauzy. The AltaRica language for Describing Concurrent Systems. *Fundamenta Informaticae*, 34(2–3):109–124, 2000.

[8] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats - A taxonomy. In *IFIP Congress Topical Sessions*, pages 91–120, 2004.

[9] M. Batteux and A. Rauzy. Stochastic Simulation of AltaRica 3.0 models. In *Proceedings of the European Safety and Reliability Conference, ESREL 2013*, Amsterdam (The Netherlands), September-October 2013. CRC Press.

[10] Romain Bernard, Jean-Jacques Aubert, Pierre Bieber, Christophe Merlini, and Sylvain Metge. Experiments in model-based safety analysis: flight controls. In Jean-Marc Faure, editor, *Proceedings of IFAC workshop on Dependable Control of Discrete Systems*, pages 43–48, Cachan, France, June 2007. Curran Associates, Inc. ISBN 9781617389948.

[11] Romain Bernard, Sylvain Metge, François Pouzolz, Pierre Bieber, Alain Griffault, and Marc Zeitoun. AltaRica Refinement for Heterogeneous Granularity Model Analysis. In *Actes du congrès Lambda-Mu'16*, page 2B, Avignon, France, October 2008. IMdR (actes électroniques).

[12] S.A. Bernardi, S. Donatelli, and J. Merseguer. From UML Sequence Diagrams and StateCharts to analyzable Petri Net models. In *In Proceedings of the Third International Workshop on Software on Performance*, 2002.

[13] Pierre Bieber, Jean-Paul Blanquart, Guy Durrieu, David Lesens, Jocelyn Lucotte, FrédéricTardy, Michel Turin, Christel Seguin, and Eric Conquet. Integration of formal fault analysis in ASSERT: Case studies and lessons learnt. In *Proceedings of 4th European Congress Embedded Real Time Software, ERTS 2008*, Toulouse, France, January 2008. SIA (electronic proceedings). code R-2008-01-2B04.

[14] M. Boiteau, Y. Dutuit, A. Rauzy, and J.-P. Signoret. The AltaRica Data-Flow Language in Use: Assessment of Production Availability of a MultiStates System. *Reliability Engineering and System Safety*, 91(7):747–755, 2006.

[15] M. Bouissou. Automated Dependability Analysis of Complex Systems with the KB3 Workbench: the Experience of EDF R&D. In *Proceedings of the International Conference on Energy and Environment*, 2005.

[16] M. Bouissou and J.L. Bon. A new formalism that combines advantages of fault-trees and Markov models: Boolean Logic Driven Markov Processes. *Reliability Engineering and System Safety*, 82:149–163, 2003.

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

[18] M. Bouissou and C. Seguin. Comparaison des langages de modélisation AltaRica et Figaro. In *15me colloque de fiabilit et maintenabilit*, Lille, France, 2006.

[19] M. Bozzano, A. Cimatti, O. Lisagor, C. Mattarei, S. Mover, M. Roveri, and S. Tonetta. Symbolic Model Checking and Safety Assessment of AltaRica Models. In *Proceedings of the 11th International Workshop on Automated Verification of Critical Systems (AVoCS 2011)*, volume 46. Electronic Communications of the EASST, 2011.

[20] Marco Bozzano, Alessandro Cimatti, and Francesco Tapparo. Symbolic fault tree analysis for reactive systems. In *Proceedings of the 5th international conference on Automated technology for verification and analysis*, pages 162–176, Berlin, Heidelberg, 2007. Springer-Verlag.

[21] Pierre-Antoine Brameret, Antoine Rauzy, and Jean-Marc Roussel. Preliminary System Safety Analysis with Limited Markov Chain Generation. In *Proceedings of 4th IFAC Workshop on Dependable Control of Discrete Systems, DCDS'2013*, pages 13–18, York, Great Britain, September 2013. International Federation of Automatic Control. ISBN: 978-3-902823-49-6, ISSN: 1474-6670.

[22] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *Computers, IEEE Transactions on*, C-35(8):677 – 691, 1986.

[23] Jean-Charles Chaudemar, Eric Bensana, Charles Castel, and Christel Seguin. AltaRica and Event-B Models for Operational Safety Analysis: Unmanned Aerial Vehicle Case Study. In *Proceedings of Workshop on Integration of Model-Based Formal Methods and Tools*, Dusseldorf, Germany, February 2009.

[24] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a Higher-order Synchronous Data-flow Language. In *ACM Fourth International Conference on Embedded Software (EMSOFT'04)*, Pisa, Italy, September 2004.

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

[26] P. David. *Contribution à l'analyse de sûreté de fonctionnement des systèmes complexes en phase de conception : application l'évaluation des missions d'un réseau de capteurs de présence humaine*. Thèse de doctorat, Université d'Orléans, 2009.

[27] P. David, V. Idasiak, and F. Kratz. Reliability study of complex physical systems using SysML. *Reliability Engineering and System Safety*, pages 431–450, 2010.

[28] Johan de Kleer. An assumption based TMS. *Artificial Intelligence*, 278(2):127–162, March 1986.

[29] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.

[30] J.B. Dugan, K.J. Sullivan, and D. Coppit. Developing a High-quality Software Tool for Fault Tree Analysis. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 222–231, 1999.

[31] Y. Dutuit, F. Innal, A. Rauzy, and J.-P. Signoret. Probabilistic assessments in relationship with Safety Integrity Levels by using Fault Trees. *Reliability Engineering and System Safety*, 93(12):1867–1876, December 2008.

[32] Y. Dutuit and A. Rauzy. Efficient Algorithms to Assess Components and Gates Importances in Fault Tree Analysis. *Reliability Engineering and System Safety*, 72(2):213–222, May 2001.

[33] S. Epstein and A. Rauzy. Open-PSA Model Exchange Format. Technical report, The Open-PSA Initiative, 2008.

[34] P. Feiler, D. Gluch, and J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, Carnegie Mellon University, 2006.

[35] P. Feiler and A.E. Rugina. Dependability Modeling with the Architecture Analysis & Design Language (AADL). Technical report, Carnegie Mellon University, 2007.

[36] S. Friedenthal, A. Moore, and R. Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. The MK/OMG Press, 2011.

[37] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. John Wiley & Sons Inc, 2004.

[38] P. Fritzson and P. Bunus. Modelica - a general object-oriented language for continuous and discrete-event system modeling and simulation. In *Simulation Symposium, 2002. Proceedings. 35th Annual*, pages 365–380, April 2002.

[39] Hauke A. L. Fuhrmann. *On the Pragmatics of Graphical Modeling*. Number 2011-1 in Kiel Computer Science Series. Department of Computer Science, May 2011. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel.

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[41] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. PEPA nets: a structured performance modelling formalism. *Performance Evaluation*, 54(2):79–104, 2003.

[42] Gregor Gössler and Joseph Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55(1-3):161–183, 2005.

[43] A. Griffault, G. Point, F. Kuntz, and A. Vincent. Symbolic computation of minimal cuts for AltaRica models. Technical report, LaBRI, Université de Bordeaux, 2011.

[44] Alain Griffault and Aymeric Vincent. The Mec 5 model-checker. In *Proceedings of the 16th International Conference on Computed Aided Verification (CAV 2004)*, volume 3114 of *Lectures Notes in Computer Science*, pages 488–491, Boston, MA, USA, July 2004. Springer Verlag.

[45] M. Güdemann and F. Ortmeier. A framework for qualitative and quantitative model-based safety analysis. In *Proceedings of 12th High Assurance System Engineering Symposium*, pages 132–141, 2010.

[46] M. Güdemann and F. Ortmeier. A Framework for Qualitative and Quantitative Model-Based Safety Analysis. In *Proceedings of the* 12^{th} *High Assurance System Engineering Symposium (HASE 2010)*, pages 132–141, 2010.

[47] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. Methods and Tools for Constraint System Architectering. In *Proceedings of the IEEE*, volume 79, pages 1305–1320, September 1991.

[48] D. Harel. StateCharts: A Visual Formalism For Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[49] A. Hatchuel and B. Weil. La théorie C-K : Fondements et usages d'une théorie unifiée de la conception. In *Actes du Colloque "Sciences de la conception"*, Lyon, France, March 2002.

[50] A. Hatchuel and B. Weil. A new approach of innovative design: an introduction to C-K theory. In *Proceedings of the International Conference on Engineering Design (ICED'03)*, Stockhholm, Sweden, August 2003.

[51] M. Hibti, T. Friedlhuber, and A. Rauzy. Overview of The Open PSA Platform. In R. Virolainen, editor, *Proceedings of International Joint Conference PSAM'11/ESREL'12*, June 2012.

[52] Sophie Humbert, Christel Seguin, Charles Castel, and Jean-Marc Bosc. Deriving Safety Software Requirements from an AltaRica System Model. In Michael D. Harrison and Mark-Alexander Sujan, editors, *Proceedings of 27th International Conference on Computer Safety, Reliability, and Security, SAFECOMP'2008*, volume 5219, pages 320–331, Newcastle upon Tyne, England, September 2008. Springer, LNCS. ISBN 978-3-540-87697-7.

[53] K. Jensen. *Coloured Petri Nets, Volume 1: Basic Concepts.* Springer-Verlag, 1992.

[54] A. Joshi, P. Binns, , and S. Vestal. Automatic generation of Fault Trees from AADL Models. In *Proceedings of the ICSE Workshop on Aerospace Software Engineering*, Minneapolis, USA, 2007.

[55] A. Joshi, S.P. Miller, M.Whalen, and Mats P.E. Heimdahl. A proposal for Model-Based Safety Analysis. In *Proceedings of the 24th Digital Avionics Systems Conference*, Washington, USA, October 2005.

[56] C. Kehren. *Motifs formels d'architectures de systèmes pour la sûreté de fonctionnement.* Thèse de doctorat, Ecole Nationale Supérieure de l'Aéronautique et de l'Espace (SUPAERO), 2005.

[57] C. Kehren, C. Seguin, P. Bieber, C. Castel, C. Bougnol, J.-P. Heckmann, and S. Metge. Architecture Patterns for Safe Design. In *AAAF 1st Complex and Safe Systems Engineering Conference (CS2E 2004)*, 2004.

[58] Minh Thang Khuu. *Contribution à l'accélération de la simulation stochastique sur des modèles AltaRica Data-Flow.* Thèse de doctorat, Université de la Méditerranée (Aix-Marseille II), 2008.

[59] L. Kloul, T. Prosvirnova, , and A. Rauzy. Modeling systems with mobile components: a comparison between AltaRica and PEPA nets. *Journal of Risk and Reliability*, 227(6):599–613, 2013.

[60] O. Lisagor, T. Kelly, and Ru Niu. Model-based safety assessment: Review of the discipline and its challenges. In *Reliability, Maintainability and Safety (ICRMS), 2011 9th International Conference on*, pages 625–632, June 2011.

[61] J.-P. Lopez-Grao, J. Merseguer, and J. Campos. From UML Activity Diagrams to Stochastic Petri Nets: Application to software performance engineering. In *In Proceedings of the Fourth International Workshop on Software and Performance*, 2004.

[62] A. Majdara and T. Wakabayashi. Component-based modeling of systems for automated Fault Tree generation. *Reliability Engineering and System Safety*, 94(6):1076–1086, 2009.

[63] M. Malhotra and K.S. Trivedi. Dependability modeling using Petri-nets. *Reliability, IEEE Transactions on*, 44(3):428–440, Sep 1995.

[64] R. Manian, J. Bechta Dugan, D. Coppit, and K.J. Sullivan. Combining various solution techniques for dynamic fault tree analysis of computer systems. In *High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International*, pages 21–28, November 1998.

[65] M. Ajmone Marsan, M. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons Inc, 1995.

[66] T. Matsuoka and M. Kobayashi. New reliability analysis methodology. *Nuclear Engineering and Design*, 98:64–78, 1988.

[67] T. Matsuoka and M. Kobayashi. The GO-FLOW reliability analysis methodology - analysis of common cause failures with uncertainty. *Nuclear Engineering and Design*, 175:205–214, 1997.

[68] T. Matsuoka, N. Mitomo, and T. Hoshi. An application of the GO-FLOW methodology – evaluation of component cooling water system for a new type of marine reactor. In *Proceedings of the 4th International Conference on Probabilistic Safety Assessment and Management (PSAM)*, volume 1, pages 221–226, New York, USA, 1998.

[69] T. Matsuoka and K. Nakagawa. An application of the GO-FLOW Methodology – a reliability analysis of automatic train control system of Shinkansen in Japan. In *Proceedings of the 4th International Conference on Probabilistic Safety Assessment and Management (PSAM)*, volume 1, pages 233–238, New York, USA, 1998.

[70] J. Merseguer, J. Campos, S. Bernardi, and S.A. Donatelli. Compositional Semantics for UML State Machines Aimed at Performance Evaluation. In *In Proceedings of the Sixth International Workshop on Discrete Event Systems*, 2002.

[71] F. Milcent, T. Prosvirnova, and A. Rauzy. Modeling network systems with AltaRica 3.0. In *Actes du congrès LambdaMu'19 (actes électroniques)*, Dijon (France), October 2014. IMdR.

[72] R. Milner. Communicating and Mobile Systems: The pi-calculus. *Cambridge University Press*, 1999.

[73] J. Noble, A. Taivalsaari, and I. Moore. *Prototype-Based Programming: Concepts, Languages and Applications*. Springer-Verlag, 1999.

[74] C. Pagetti. *Extension temps réel du langage AltaRica*. Thèse de doctorat, École Centrale de Nantes et de l'Université de Nantes, 2004.

[75] Y. Papadopoulos and M. Maruhn. Model-Based Synthesis of Fault Trees from Matlab-Simulink Models. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks*, DSN '01, pages 77–82, Washington, DC, USA, 2001. IEEE Computer Society.

[76] Y. Papadopoulos, M. Walker, D. Parker, E. Rude, R. Hamann, A. Uhlig, U. Gratz, and R. Lien. Engineering failure analysis and design optimization with HiP-HOPS. *Engineering Failure Analysis*, 18:590–608, 2011.

[77] Y. Papadopoulos, M. Walker, M.-O. Reiser, M. Weber, D. Chen, M. Törngren, David Servat, A. Abele, F. Stappert, H. Lonn, L. Berntsson, Rolf Johansson, F. Tagliabo, S. Torchiaro, and Anders Sandberg. Automatic Allocation of Safety Integrity Levels. In *Proceedings of the 1st Workshop on Critical Automotive Applications: Robustness & Safety*, CARS '10, pages 7–10, New York, USA, 2010. ACM.

[78] A. Pasquini, Y. Papadopoulos, and J. McDermid. Hierarchically performed hazard origin and propagation studies. *Computer Safety, Reliability and Security*, 1698 of LNCS:688–688, 1999.

[79] B. Perrot, T. Prosvirnova, A. Rauzy, J.-P. Sahut d'Izarn, and R. Schoening. Expériences de couplages de modèles AltaRica avec des interfaces métiers. In E. Fadier, editor, *Actes du congrès LambdaMu'17 (actes électroniques)*. IMdR, October 2010.

[80] G. Point and A. Rauzy. AltaRica: Constraint automata as a description language. *Journal Européen des Systèmes Automatisés*, 33(8–9):1033–1052, 1999.

[81] Adrian Pop and Peter Fritzson. The Modelica Standard Library as an Ontology for Modeling and Simulation of Physical Systems. Technical report, Linkping University, PELAB - Programming Environment Laboratory, 2004.

[82] T. Prosvirnova, M. Batteux, P.-A. Brameret, A. Cherfi, T. Friedlhuber, J.-M. Roussel, and A. Rauzy. The AltaRica 3.0 project for Model-Based Safety Assessment. In *Proceedings of 4th IFAC Workshop on Dependable Control of Discrete Systems, DCDS 2013*, York (Great Britain), September 2013. IFAC.

[83] T. Prosvirnova, M. Batteux, A. Maarouf, and A. Rauzy. GraphXica: a Language for Graphical Animation of models. In *Proceedings of the European Safety and Reliability conference, ESREL 2013*, Amsterdam (The Netherlands), September-October 2013. CRC Press.

[84] T. Prosvirnova and A. Rauzy. Guarded Transition Systems: Pivot Modelling Formalism For Safety Analysis. In J.F. Barbet, editor, *Actes du Congrès Lambda-Mu 18*, Octobre 2012.

[85] Xavier Quayzin and Emmanuel Arbaretier. Performance Modeling of a Surveillance Mission. In *Proceedings of the Annual Reliability and Maintainability Symposium, RAMS'2009*, pages 206–211, Fort Worth, Texas, USA, January 2009. IEEE. ISBN 978-1-4244-2508-2.

[86] A. Rauzy. New Algorithms for Fault Trees Analysis. *Reliability Engineering and System Safety*, 05(59):203–211, 1993.

[87] A. Rauzy. Mathematical Foundation of Minimal Cutsets. *IEEE Transactions on Reliability*, 50(4):389–396, december 2001.

[88] A. Rauzy. Mode Automata and their Compilation into Fault Trees. *Reliability Engineering and System Safety*, 78(1):1–12, 2002.

[89] A. Rauzy. An experimental study on iterative methods to compute transient solutions of large Markov models. *Reliability Engineering & System Safety*, 86(1):105–115, 2004.

[90] A. Rauzy. Guarded Transition Systems: a new States/Events Formalism for Reliability Studies. *Journal of Risk and Reliability*, 222(4):495–505, 2008.

[91] A. Rauzy. Anatomy of an Efficient Fault Tree Assessment Engine. In R. Virolainen, editor, *Proceedings of International Joint Conference PSAM'11/ESREL'12*, Helsinki, Finland, June 2012.

[92] A. Rauzy. AltaRica Data-Flow language specification. Technical report, Ecole Polytechnique, 2013. version 2.1.

[93] Antoine Rauzy. BDD for Reliability Studies. In K.B. Misra, editor, *Handbook of Performability Engineering*, pages 381–396. Elsevier, 2008. ISBN 978-1-84800-130-5.

[94] D. Riera, F. Milcent, J .Parisot, and E. Clement. Dynamic modeling for dependability and safety evaluation: an advance for the analysis of complex systems, Octobre 2012.

[95] A.E. Rugina, K. Kanoun, and M. KAANICHE. The ADAPT Tool: from AADL Architectural Models to Stochastic Petri Nets through Model Transformation. In *7th European Dependable Computing Conference*, Kaunas, Lithuanie, 2008.

[96] L. Sagaspe. *Allocation sûre dans les systèmes aéronautiques : Modélisation, Vérification et Génération*. Thèse de doctorat, Université BORDEAUX I, 2008.

[97] Laurent Sagaspe and Pierre Bieber. Constraint-Based Design and Allocation of Shared Avionics Resources. In *Proceedings of 26th AIAA-IEEE Digital Avionics Systems Conference*, pages 2.A.5–1–2.A.5–10, Dallas, Texas, USA, October 2007. IEEE.

[98] Manel Sghairi, Agnan De Bonneval, Yves Crouzet, Yves Aubert, Patrice Brot, and Youssef Laarouchi. Distributed and reconfigurable architecture for flight control system. In *Proceedings of 28th Digital Avionics Systems Conference (DASC'09)*, pages 6.B.2–1–6.B.2–10, Orlando, Florida, Etats-Unis, October 2009.

[99] M.L. Shooman. The Equivalence of Reliability Diagrams and Fault-Tree Analysis. *Reliability, IEEE Transactions on*, R-19(2):74–75, May 1970.

[100] J.-P. Signoret. Dependability & Safety Modeling and calculation: Petri Nets. In *Proceeding of the 2nd IFAC Workshop on Dependable Control of Descrete Systems, DCDS 2009*, Bari, Italy, June 2009.

[101] William J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[102] K.J. Sullivan, J.B. Dugan, , and D. Coppit. The Galileo Fault Tree Analysis Tool. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, 1999.

[103] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

[104] Leslie G. Valiant. The Complexity of Enumeration and Reliability Problems. *SIAM Journal of Computing*, 8(3):410–421, 1979.

[105] J.-L. Voirin. Methods and Tools for Constraint System Architectering. In *Proceedings of the 18th International Symposium of the International Council on System Engineering (INCOSE 2008)*, pages 775–789. Curran Associates Incorporated, 2008, June 2008.

[106] M. Walker and Y.Papadopoulos. Qualitative temporal analysis: Towards a full implementation of the Fault Tree Handbook. *Control Engineering Practice*, 17:1115 – 1125, 2009.

[107] John X. Wang and Marvin L. Roush. *What Every Engineer Should Know About Risk Engineering and Management*. CRC Press, 2000. ISBN 0824793013.

[108] S. White and D. Miers. *BPMN Modeling and Reference Guide: Understanding and Using BPMN*. Future Strategies Inc., 2008.

[109] J. Xiang, K. Yanoo, Y. Maeno, and K. Tadano. Automatic Synthesis of Static Fault Trees from System Models. In *Conference on Secure Software Integration and Reliability Improvement*, pages 127–136, 2011.

AltaRica 3.0 : une approche orientée modèles pour la Sûreté de Fonctionnement

Thèse de Doctorat en Informatique — Tatiana Prosvirnova

Résumé

La sûreté de fonctionnement des systèmes est un domaine en plein essor. Les ingénieurs fiabilistes ont mis au point diverses méthodes d'analyse du risque qui sont aujourd'hui bien maîtrisées: les Arbres de Défaillance ou les Arbres d'Evénements. Des algorithmes efficaces et des outils performants sont disponibles pour évaluer les modèles. Ces formalismes ont cependant comme inconvénient majeur d'être éloignés des descriptions fonctionnelles des systèmes. Il en résulte un décalage, toujours dangereux, entre les spécifications techniques du système étudié et les modèles utilisés par les fiabilistes. Maintenir ces derniers tout au long du cycle de vie des produits est donc une tâche difficile, coûteuse et susceptible de comporter des erreurs.

Le langage AltaRica Data-Flow a été créé pour pallier ce problème. AltaRica Data-Flow est un langage de modélisation de haut niveau permettant de décrire des composants sous forme d'automates d'états finis, de créer des bibliothèques de modèles de composants et d'assembler ces modèles en des hiérarchies. Il a été choisi comme langage support de plusieurs ateliers logiciels utilisés dans l'industrie.

La thèse porte sur la nouvelle version du langage AltaRica 3.0. Elle améliore AltaRica Data-Flow selon deux axes: son modèle d'exécution est basé sur les Systèmes de Transitions Gardées, ce qui permet de modéliser les systèmes bouclés et les composants avec les flux bidirectionnels; nouvelles constructions pour structurer les modèles, qui proviennent des langages orientés prototype, sont introduites. La thèse comporte une partie formelle décrivant les nouvelles constructions structurelles et précisant la sémantique du langage, une partie algorithmique expliquant la compilation des modèles AltaRica 3.0 vers les Arbres de Défaillance et la mise en oeuvre des algorithmes dans un prototype.

* * *

AltaRica 3.0: a Model-Based approach for Safety Analyses

Abstract

The Model-Based approach for safety and reliability analyses is gradually wining the trust of engineers but is still an active domain of research. Safety engineers master "traditional" risk modeling formalisms, such as Fault Trees and Event Trees. Efficient algorithms and tools are available. However, despite of their qualities, these formalisms share a major drawback: models are far from the specifications of the systems under study. As a consequence, models are hard to design and to maintain throughout the life cycle of systems. A small change in the specifications may require a complete revisiting of the safety models, which is both resource consuming and error prone.

The high level modeling language AltaRica Data-Flow has been created to tackle this problem. AltaRica Data-Flow models are made of hierarchies of reusable components. Graphical representations are associated with components, making models visually very close to Process and Instrumentation Diagrams. AltaRica Data-Flow is at the core of several Integrated Modeling and Simulation Environments used in industry.

AltaRica 3.0 is a new version of the language. It improves AltaRica Data-Flow into two directions: its semantics is based on the new underlying mathematical model Guarded Transition Systems (GTS), which makes it possible to handle systems with instant loops and to define acausal components, i.e. components for which the input and output flows are decided at run time; it provides new constructs to structure models, coming from prototype-oriented modeling languages. The thesis includes a formal part describing in detail the new structural constructs and the semantics of the language, an algorithmic part explaining the compilation of AltaRica 3.0 models into Fault Trees and an implementation of the algorithms in a prototype.

LIX Ecole Polytechnique – Novembre 2014