Synthesizing Fault-Tolerant Programs from Deontic Logic Specifications

SYNTHESIZING FAULT-TOLERANT PROGRAMS FROM DEONTIC LOGIC SPECIFICATIONS

BY
RAMIRO ADRIAN DEMASI, Lic.

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE
AND THE SCHOOL OF GRADUATE STUDIES
OF MCMASTER UNIVERSITY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Doctor of Philosophy (2014) McMaster University
(Computer Science) Hamilton, Ontario, Canada

TITLE:	Synthesizing Fault-Tolerant Programs from Deontic Logic Specifications
AUTHOR:	Ramiro Adrian Demasi, Lic. Universidad Nacional de Río Cuarto, Argentina
SUPERVISORS:	Dr. Thomas S.E. Maibaum (McMaster University) and Dr. Pablo F. Castro (Universidad Nacional de Río Cuarto, Argentina)
NUMBER OF PAGES:	xiv, 163

To my mom and dad

Abstract

This dissertation concentrates on the problem of synthesizing fault-tolerant components from specifications, i.e., the problem of automatically constructing a fault-tolerant component implementation from a logical specification of the component, and the system's required level of fault-tolerance. In our approach, the logical specification of the component is given in dCTL-, a fragment of a branching time temporal logic with deontic operators, especially designed for fault-tolerant component specification. Deontic logics have proved to be useful for reasoning about legal and moral systems, where the situation is more or less similar to fault-tolerance: there exists a set of rules that states what the normal behaviours or scenarios are. Violations arise when these rules are not followed and, as a consequence, some actions must be performed to return to a normal or desirable state.

As a black-box overview, our synthesis algorithm takes the component specification and a user-defined level of fault-tolerance (masking, nonmasking, or failsafe), and automatically determines whether a component with the required fault-tolerance is realizable. Moreover, if the answer is positive, then the algorithm produces such a fault-tolerant implementation. Our technique for synthesis is based on the use of (bi)simulation algorithms for capturing different fault-tolerance classes, and the extension of a synthesis algorithm for CTL to cope with dCTL- specifications.

Some case studies are provided throughout this thesis to illustrate how the ideas described below can be applied in practice. Moreover, we have implemented a tool called dCTL Synthesizer (syntdctl) which was used to synthesize automatically well-known fault-tolerant examples.

Acknowledgements

First of all, I would like to express my deepest gratitude to my supervisor Tom Maibaum, for his advice, guidance, and encouragement throughout the course of this work. It has been a pleasure working with him.

I am also deeply grateful to my other advisor Pablo Castro, for investing many hours, in making sure that I did things right. He has been a perfect partner with whom working long hours is extremely productive and fun.

Another person I would like to thank is Nazareno Aguirre who has taught me a lot of things and who has given me a lot of valuable advice during the last seven years. Naza was also one of the people who motivated me to come to do a PhD in Canada.

I am particularly grateful to my friend Valentin, for being a very supportive friend during my stay in Canada. I am also deeply indebted with numerous friends not only from Canada but also from Argentina who endured this long process with me, always offering support and love.

I would like to thank the members of my PhD committee, Dr. Mark Lawford and Dr. Emil Sekerinski, who have supported me with their valuable comments. I also want to thank Dr. Sandeep Kulkarni for his valuable comments and suggestions for future extensions of my work.

Financial support during my graduate studies has been generously provided by a Fellowship from IBM Canada in the last three years, the Department of Computing and Software at McMaster University for the first two years through a Graduate Scholarship and NSERC for support during the same period.

Above all I would like to thank my family, especially my parents Raúl and Monica and sisters Romina, Cintia, and Sofi, for all their support and love.

Saving the best for last: Flor, thank you so much for everything, your support, your love, your patience, everything. Without her this PhD would not have been

possible.

Contents

List of Tables

List of Figures

Declaration of Academic Achievement

Ramiro Demasi, Pablo F. Castro, T.S.E. Maibaum, and Nazareno Aguirre (2013a). Characterizing Fault-Tolerant Systems by Means of Simulation Relations. In E. B. Johnsen and L. Petre (Eds.), *Integrated Formal Methods, 10th International Conference, IFM 2013*, Volume 7940 of *Lecture Notes in Computer Science*, pp. 426-440, 2013. Springer.

Ramiro Demasi, Pablo F. Castro, T.S.E. Maibaum, and Nazareno Aguirre (2013b). Synthesizing Masking Fault-Tolerant Systems from Deontic Specifications. In D. V. Hung and M. Ogawa (Eds.), *Automated Technology for Verification and Analysis, 11th International Symposium, ATVA 2013*, Volume 8172 of *Lecture Notes in Computer Science*, pp. 163-177, 2013. Springer.

Ramiro Demasi (2013). Synthesizing Fault-Tolerant Programs from Deontic Logic Specifications. In *28th IEEE/ACM International Conference on Automated Software Engineering - Doctoral Symposium, ASE 2013*, pp. 750-753, 2013. IEEE.

Ramiro Demasi, Pablo F. Castro, T. S. E. Maibaum, and Nazareno Aguirre (2014). Simulation Relations for Fault-Tolerance. *Submitted for journal publication.*

The results described in this dissertation have been submitted for publication in conferences. So far, three papers [Demasi et al., 2013a], [Demasi et al., 2013b], and [Demasi, 2013] have been accepted. Additionally, [Demasi et al., 2014] has been submitted to a journal. Each of these papers is co-authored and I am the lead author

for each. Overall, I conceived of each paper with my supervisors (Dr. Tom Maibaum and Dr. Pablo Castro) and Dr. Nazareno Aguirre (my Licentiate's supervisor from Universidad Nacional de Río Cuarto, Argentina). In more details, the general idea (i.e., synthesis of fault-tolerant programs from deontic logic specifications) of this dissertation was conceptualized and conducted by the PhD candidate (Ramiro Demasi). Specifically, my supervisor Dr. Pablo Castro suggested that I characterize the different levels of fault-tolerance (masking, nonmasking, and failsafe) by means of simulation/bisimulation relations. I have contributed with the technical definitions, proofs, and algorithms presented in Chapter 3 and 4 and those given in the aforementioned papers. Additionally, I have contributed with the adaptation and extension of the tableau-based synthesis from CTL specifications, to deal with the additional deontic modalities, and bisimulation algorithms. These results have been possible after valuable comments and suggestions by Dr. Pablo Castro. The main idea of using deontic logic for reasoning about fault-tolerant systems is due to my supervisor Dr. Tom Maibaum, who guided me to find these formalisms and the corresponding intuitions in this topic. My supervisors and Dr. Aguirre also contributed to the analysis during ongoing iterative cycles of interpretation of the general problem that led to the development of the final synthesis method. I alone was responsible for the implementation of the syntdctl tool and also for the specification and running the experiments for all case studies. Finally, I drafted all chapters and each of my supervisors provided comments and suggestions that were incorporated into revisions.

Chapter 1

Introduction

The increasing demand for highly dependable and constantly available systems has focused attention on providing strong guarantees for software robustness, understood as the ability of software to continue operating in an acceptable way despite erroneous behavior during its execution or the existence of an uncooperative environment; this is particularly true for safety-critical systems. Some examples of such safety-critical systems include software for medical devices and software controllers in the avionics and automotive industries. Unfortunately, there are many examples of safety-critical systems which, due to software systems malfunctioning, ended up being big catastrophic failures, such as the Ariane 5, the radiation therapy machine Therac-25, and the Denver Airport baggage handling system. These and many other examples of catastrophic failures are described in [Peterson, 1996]. On June 4, 1996 the Ariane 5 rocket, launched by the European Space Agency, exploded just forty seconds after initiation of the flight sequence, due to a software problem. Specifically, a component raised an operand error exception while converting a 64-bit floating point number to a 16-bit integer, where no specific exception handler was defined; then, the uncaught exception caused the termination of the system and consequently, an estimated loss of $500 million. The Therac-25 radiation-treatment machine for cancer treatment injured and even killed several patients (six accidents between 1985 and 1987) by administering massive radiation overdoses. Denver Airport planned to automate the handling of luggage through the entire airport, using software controlled conveyor belts; the system never worked well; bugs delayed the airport's opening for months. After 10 years of repetitive failures, it has been abandoned, and consequently more

than $15 millions were wasted. In general, the causes of these and other failures in safety-critical systems is due to different factors, such as poor management of requirements, inconsistency between the different construction phases, like requirements and implementation phases, limited or no use of verification and validation techniques, etc. As we observe, safety-critical systems are subject to a variety of potential failures that can corrupt or degrade their performance; so, being able to reason about computer systems behavior in the presence of potential failures in order to provide strong guarantees for software correctness has gained considerable attention.

The field of *fault-tolerance* is concerned with providing techniques that can be used to guarantee reliability and availability of critical services as well as application execution. This includes specific mechanisms for achieving fault-tolerance, as well as for appropriately modeling fault-tolerant systems, and expressing and reasoning about fault-tolerant behaviors. Some examples of traditional techniques employed to deal with fault-tolerance are: *component replication*, *N-version programming*, *exception mechanisms*, *transactions*, etc. All of these techniques can add confidence to safety-critical systems about their capability for dealing with faults (standard references to the field of fault-tolerance are [Lee and Anderson, 1990; Avizienis, 1995; Prasetya and Swierstra, 2005; Siewiorek and Swarz, 1998; Torres-Pomales, 2000]). However, these techniques are mainly for the implementation phase and not for the design phase.

Several approaches have been proposed to deal with fault-tolerance in formal settings, with the main aim of mathematically *proving* that a given system effectively tolerates faults, and thus implements reliable software. Some examples are the use of program transformation [Kulkarni and Arora, 2000; Kulkarni and Ebnenasir, 2003, 2004; Ebnenasir et al., 2008; Bonakdarpour et al., 2012], process algebra based approaches [Janowski, 1995, 1997], specification languages (e.g., Alloy [Kang and Jackson, 2008], TLA+ [Lamport and Merz, 1994], and Event-B [Yadav and Butler, 2009]), etc. Related to the last, recently some researchers (e.g., [Carmo and Jones, 1996a; Khosla and Maibaum, 1987; Kent et al., 1991; Khosla, 1989; Lomuscio and Sergot, 2004; Fiadeiro and Maibaum, 1991]) have pointed out that *deontic logic*, a variation of logic advocated for the study of norms, is useful for reasoning about fault-tolerant systems. Deontic logics have proved to be useful for reasoning about legal and moral systems, where the situation is more or less similar to fault-tolerance: there exists a set of rules that states what the normal behaviours or scenarios are. Violations

arise when these rules are not followed and, as a consequence, some actions must be performed to return to a normal or desirable state.

In the last three decades, automated verification (particularly model checking) is probably the most exciting and successful advance of formal methods applied to hardware and software development. In order to achieve automatic correctness in system design, two different approaches have been applied: correct-by-verification and correct-by-construction. Recently, formal approaches involving *model checking*, applied to fault-tolerance, have been proposed (e.g., see [Bernardeschi et al., 2002; Schneider et al., 1998; Yokogawa et al., 2001]). In these approaches, temporal logics are employed to capture fault-tolerance properties of reactive systems, and then *model checking* algorithms are used to automatically verify that these properties hold for a given system. Since model checking provides fully automated analysis (for finite systems), and counterexamples are generated when a property does not hold (which is extremely helpful in finding the source of the problem in the system), model checking based approaches to fault-tolerance provide significant benefits over other semi-automated or manual formal approaches. However, the languages employed for the description of systems and system properties in model checking do not provide a built-in way of distinguishing between normal and abnormal behaviors. Thus, when capturing fault-tolerant systems, and expressing fault-tolerance properties, the specifier needs to *encode* in some suitable way the faults and their consequences. This makes formulas longer and more difficult to understand, which has an obvious negative impact on the analysis, since the performance of model checking algorithms depends on the length of the formula being analyzed; moreover, the counterexamples generated are harder to follow. On the other hand, but with less emphasis, approaches for automatically *synthesizing* programs, in particular fault-tolerant ones, have also been studied [Attie et al., 2004; Bonakdarpour et al., 2012; Kulkarni and Arora, 2000; Kulkarni and Ebnenasir, 2004].

In this thesis, we study the problem of automatically synthesizing fault-tolerant systems from logical specifications, i.e., the problem of automatically constructing a fault-tolerant component implementation from a logical specification of the component, and the system's required level of fault-tolerance. In our approach, the logical specification of the component is given in dCTL-, a fragment of a branching time temporal logic with deontic operators, especially designed for fault-tolerant component

specification. As a black-box overview, our synthesis algorithm takes the component specification and a user-defined level of fault-tolerance (masking, nonmasking, or failsafe, see below), and automatically determines whether a component with the required fault-tolerance level is realizable. Moreover, if the answer is positive, then the algorithm produces such a fault-tolerant implementation. Our technique for synthesis is based on the use of (bi)simulation algorithms for capturing the different fault-tolerance classes, and the extension of a synthesis algorithm for CTL to cope with dCTL- specifications. Some case studies are provided throughout this thesis to illustrate how the ideas described below can be applied in practice. Moreover, we have implemented a tool, called Synthesizer of dCTL- (syntdctl), which was used to synthesize automatically solutions to well-known fault-tolerant examples.

1.1 Fault-Tolerant Systems

Fault-tolerance is the ability of a system to perform its function correctly even subject to the occurrence of faults. The objective of fault-tolerance is to increase the dependability of a system. Generally speaking, a fault-tolerant system is able to mitigate the occurrence of faults in order to guarantee behaviors that will not cause critical failures of the application. Nowadays, most safety safety-critical systems require full fault-tolerance, i.e., the system continues operating in the presence of faults, perhaps for a limited period, with no important loss of functionality or performance observed by the user. In the worst case, the system fails safely, i.e., in a state that does not cause a disaster. However, in practice, graceful degradation is an alternative approach used in the face of faults, where the system continues to operate in the presence of faults, accepting a partial degradation of functionality or performance during recovery or repair. For example, considering the Anti-lock braking system (ABS) in a car, if there is a situation in which a sensor is broken, then the brake should continue to work under manual control even under the malfunction of the sensor.

According to [Chou, 1997; Torres-Pomales, 2000; Guelfi et al., 2007], software faults are the root cause in a high percentage of operational system failures. The consequences of these failures depend on the application and on the particular characteristics of the faults. The repercussions of these can range from minor problem (e.g., having to restart a personal computer) to catastrophic events (e.g., software

in an aircraft that prevents the pilot from recovering from an input error). In the context of business, the consequences caused by operational failures are related to loss of potential customers, lower sales, higher warranty repair costs, and losses due to legal actions from the people affected by the failures.

Fault-tolerance has been extensively studied in the literature: [Avizienis et al., 2004a] gives an exhaustive list of the basic concepts and terminology of fault-tolerance, and [Gärtner, 1999a] formalizes the important underlying notions of fault-tolerance.

In the following subsections we explore the main concepts of fault-tolerance, as well as different tactics to achieve dependability, and finally we give a brief description of software and hardware fault-tolerance techniques.

1.1.1 Faults, Errors, and Failures

A computer system may be affected by events that can menace its ability to deliver desirable and correct services. A commonly occuring cause is when engineers or developers of systems have introduced unintended defects or bugs during the construction phase. Another common factor is related to hardware defects that may threaten the computer system's functionality, for example unexpected events produced in noisy environments. These factors that can generate unexpected malfunctions in computer systems are designated as *faults*. In short, a *fault* is either a hardware defect or a software/programming mistake (i.e., a bug).

An *error* is an undesired state of a system, which is a manifestation of a fault. Notice that, an undesired state is reached in a computer system only when the fault has been *activated*; this means that the fault itself may not cause a malfunction of the system. For example, suppose that a fault, like a programming mistake, exists in a certain area of the memory that is not accessed (i.e., the fault is *dormant*); in this sense, there is no harm to be expected from the behavior of the system as a result of this fault. Nevertheless, in case we access that area of memory during the execution of the system, the fault is activated and we observe an error. Consequently, the result obtained may be used for further computations that can affect the expected behavior of the system, violating its specification, i.e., the required service is not delivered. The use of the computed faulty value for further computation is referred to as *propagation*. Finally, a *failure* is the consequence of error propagation to the output of the computer system, i.e, there is an observable deviation of the behaviour

of the system from what is prescribed by its specification. Therefore, the main goal of fault-tolerance is to avoid system failure in the presence of faults. We have explained these basic concepts following the terminology of fault-tolerance defined in [Avizienis et al., 2004b].

1.1.2 Levels of Fault-Tolerance

A fault-tolerant system is able to cope with situations where a subset of its components are affected by the occurrence of faults. Depending on the chosen type of fault-tolerance, the system can deal with the faults in different ways. In general, three levels of fault-tolerance are considered: *masking*, *nonmasking* and *failsafe*. *Masking fault-tolerance* corresponds to the case in which the system may completely mask the faults, not allowing these to have any observable consequences for the users; *nonmasking fault-tolerance* corresponds to the case in which, after a fault occurs, the system may undergo some process, observable by users, to eventually take the system back to a "good" behavior; finally, *failsafe fault-tolerance* corresponds to the case in which the system may react to a fault by switching to a behavior that is safe, but in which the system is restricted in its capacity.

As argued by the author in [Gärtner, 1999a], these fault-tolerance properties can be classified in term of the varying satisfaction of the system's safety and liveness properties in the presence of faults:

- *Masking fault-tolerance*: the program continuously satisfies the safety and liveness specifications, even in the presence of faults, i.e., the program never violates the safety part of the specification and, in the case that a fault occurs, it eventually recovers to its normal behavior. Some examples of systems which require masking fault-tolerance are those based on consensus, agreement, voting, or commitment.

- *Failsafe fault-tolerance*: the requirement for failsafe fault-tolerance is that only the safety part is guaranteed, but not necessarily the liveness part. A standard example of this is a nuclear power plants where, subject to the occurrence of faults, we need to ensure that the system goes into a safe state (e.g., shut the system down), where perhaps some liveness properties are not preserved, but the system is kept in a safe state.

- *Nonmasking fault-tolerance*: the liveness part is always guaranteed but the safety part is only eventually respected. Intuitively, this type of fault-tolerance allows the program to violate the safety specification while it is recovering to the normal behavior. Systems based on reset, checkpointing/recovery, or exception handling typically require nonmasking fault-tolerance.

These fault-tolerance properties capture the fault-tolerance requirements of extant computing systems. In [Arora and Gouda, 1993], the authors discuss how these fault-tolerance properties capture the requirements in distributed systems, networks, circuits, database management, etc.

1.1.3 Approaches to Dependability

In order to deal with faults, developers have been applying various techniques over the last 50 years. These methods can be grouped into the following four classes, as explained in [Torres-Pomales, 2000]:

- *Fault Avoidance/Prevention:* the main goal is to prevent and reduce the introduction and the occurrence of faults during software construction. A rigorous software development process is necessary to produce software of good quality. In particular, formal methods provide an appealing approach to produce software without faults; theories and languages arising from mathematics are used to prove mathematically that software is free of faults. If despite fault avoidance efforts, faults are created, then fault removal is needed.

- *Fault Removal:* the aim is to detect and remove existing faults during software verification and validation. Exhaustive and rigorous testing is the usual technique to attain this, which can be done in several different ways to discover faults during the development process and therefore delete them from the design and the implementation phase. Other common techniques are formal inspection and formal design proofs. Particularly, model checking is a successful method that can be used to check that the implementation corresponds to the specification of the system. Fault removal is imperfect, so fault forecasting and fault-tolerance are needed.

- *Fault Forecasting:* also called as software reliability measurement by Lyu in [Lyu, 1996]. It provides information during the validation phase to enhance dependability by estimating the present number, future incident, and consequence of faults. This method can indicate the need for additional testing or for applying fault-tolerance.

- *Fault-Tolerance:* the main idea is to tolerate faults that remain in the system after its development, preventing system failure and delivering the expected functionality even in the presence of faults. In order to accomplish it, the system should be able to detect the occurrence of errors and, eventually, to recover from those errors in the system. There are several fault-tolerance techniques (see Section 1.1.4) which help to reduce the risks of software design faults and thus enhance the dependability of the system.

As we observed, testing techniques and formal methods play an important role in the above techniques. However, system testing can never be exhaustive and remove all potential faults; as Dijkstra in [Dijkstra, 1972] argued, testing can show the presence, but not the absence of faults. In the case of formal methods, these techniques are hard to apply to large and complex systems. In addition, we can achieve a system which is free of faults, but the system will always be exposed to faults from a malicious environment, or even from the operating system and possible malfunctions of hardware. Therefore, fault-tolerance techniques are needed in order to provide a guarantee that safety-critical systems continue working in an acceptable way, even subject to the occurrence of faults.

1.1.4 Fault-Tolerance Techniques

Several fault-tolerance techniques have been developed in order to improve the capability of a system to deal with faults. These are commonly divided into fault-tolerance hardware and software techniques, depending on whether they are used at the hardware level or the software level. We briefly review both approaches.

Hardware fault-tolerance techniques

The main approach to attaining hardware fault-tolerance is by using additional hardware, better known as *redundancy*. In general, hardware techniques can be classified

into *static*, *dynamic*, and *hybrid* redundancy.

Static redundancy is characterized by using redundant components inside a system in order to hide the effects of faults. A typical example is the use of Triple Modular Redundancy (TMR), in which three identical subcomponents perform a process and the result of all of them are compared by a majority voting system producing a single output. If one of them differs from the other two that output is masked out by the other two subcomponents. In short, these techniques prevent the faults from resulting in errors using fault masking to hide the effects of faults. Moreover, fault-tolerance is achieved without requiring any system or operator action. Finally, these techniques do neither detect faults nor change the configuration of the hardware dynamically. On the contrary, dynamic redundancy achieves fault-tolerance by detecting the existence of faults and performing some action to remove the faulty parts. It also requires the system to be reconfigured to tolerate faults. Specifically, three steps are performed in these techniques in an attempt to achieve fault-tolerance: fault detection, fault location, and fault recovery. There are many dynamic mechanisms like duplication with comparison and the pair-and-a-spare technique. In the former, two modules perform the same computations in parallel and compare the results. In the case that the two results disagree, then an error message is generated and subsequently some recovery actions are performed in order to take the system back to an acceptable state. In the latter, two modules are operated in parallel at all times and their results are compared to provide the error protection capability. The error signal from the comparison is used to initiate the reconfiguration process (switch) that removes faulty modules and replaces them with spares. Finally, hybrid hardware redundancy combines the attractive features of both static and dynamic techniques: fault masking, fault detection, fault location, and fault recovery. Most hybrid redundancy techniques are based on the concept of N-modular redundancy (NMR) with spare. The idea is to provide N modules arranged in a voting configuration. Moreover, spares are provided to replace failed modules. The main advantage of NMR with spares is that voting can be restored after a fault has occurred.

Hardware fault-tolerance has been investigated deeply and successfully applied in aeronautics, nuclear applications, aerospace systems, healthcare, telecommunications and transportation industries. A good reference is [Siewiorek and Swarz, 1998].

Software fault-tolerance techniques

Software fault-tolerance techniques are used to build software capable of tolerating faults. Following [Torres-Pomales, 2000], they can be classified into two classes:

- *Single version fault-tolerance*: this class focuses on improving the fault-tolerance of a single piece of software by adding mechanisms into the design. Specifically, targeting the detection, the containment, and recovery from errors caused by the activation of design faults. There are several techniques that use a single-version of some software. As explained in [Torres-Pomales, 2000], decomposing software into several components or pieces which are independent to some degree is important to avoid the propagation of errors from one part of the system to other parts, or possibly all of the system. Here, it is important to use techniques that restrict the propagation of errors from one component to others when designing the architecture of the software. For example, the *system closure* technique is based on a principle that no action is permitted unless it is explicitly authorized [Denning, 1976]. Then, when an error is detected, the system reacts disabling any valid actions to avoid error propagation. *Atomic actions* [Lee and Anderson, 1990] is another approach in which a group of components interact with each other and there is no communication among these components and with the rest of the system during this atomic activity. There are two possible outcomes of an atomic action: it either terminates normally or it is aborted upon a detection of a fault. In the first case, its results are complete and committed. In the second case, it is known in advance that only the participating components can be affected; therefore it is possible to isolate the errors to the participating components. Another common technique for tolerating software design faults is the use of *exception mechanisms*, that is the interruption of normal operation to respond to the occurrence of unexpected events (exceptions) requiring special processing - often changing the normal flow of program execution. In [Randell and Xu, 1994], the authors list three classes of exception triggering events for a software component: interface exceptions (a module raises an exception immediately upon detection of invalid service request), local exceptions (a module raises an exception when its fault detection mechanism detects a fault), and failure exceptions (a module raises an exception to signal that its recovery mechanism is enabled to recover successfully).

Many programming languages provide mechanisms for exception handling. Well-known examples are: *Ada*, *Java*, *C++*, *Eiffel*, *ML*, and *Smalltalk*. Other techniques used are *checkpoints*, classified into static and dynamic. On the one hand, static checkpoint stores in memory information of a single snapshot of the system at the beginning of the program execution. If a fault is detected, then the system goes back to this state and restarts the execution of the program from the beginning. On the other hand, dynamic checkpoints are generated dynamically at different points during the execution. In the case that a fault is detected, then the system goes back to the last checkpoint and continues the program execution. Note that the checks for fault detection need to be inserted in the code and be executed prior to when the checkpoints are generated. Finally, *recovery mechanisms* are used once a fault is detected and contained, a system attempts to recover from the faulty state and reach a safe or correct state. If fault detection mechanisms are implemented correctly, then the consequences of the faults are enclosed within an appropriate set of modules at the time of fault detection. Most importantly, knowledge of the fault containment region is indispensable for the design of an efficient fault recovery mechanism.

- *Multiversion fault-tolerance*: is based on the use of two or more versions (or variants) of a piece of software, executed either in sequence or in parallel to prevent system failures. The main idea is that components are built differently through different designer teams, different programming languages or different algorithms can be used to maximize the probability that all the versions do not have common faults. Therefore, if one version fails on particular input, at least one of the alternate versions should be able to provide an appropriate output. *N-version programming* [Avizienis, 1995] and *Recovery blocks* (RB) are two popular examples of these techniques. The former, is a technique in which several versions of some software are produced to satisfy the same specification, and then the output of the system is decided considering all the outputs from the execution of a task. This decision is based on a voting mechanism which determines the result based on majority voting or some other selection rules. There exist diverse variations of this approach, essentially changing the way in which the output is selected, and also combining this technique with single-version methods. The latter, RB uses multiple alternates (backups) to perform

the same function. One task is selected as primary and the others as secondary. Initially, the primary task executes a particular function. After its completion, an acceptance test is applied to check its result. In the case that the output is not acceptable, then a secondary task executes the same function at the state in which was invoked the primary task. This process continues until an acceptable result is achieved or the deadline of the task is missed. See [Torres-Pomales, 2000] for a more exhaustive list of methods that have been used in practice.

These techniques have to be complemented with an important aspect of fault-tolerance: the detection of errors. The aim is to define efficient mechanisms in order to detect errors during the execution of the system. In the literature, we found that the most usual forms of implementation of error detection are the following: error detecting codes, duplexing and comparison, timing and execution checks, reasonableness checks, and structural checks. The latter three checks are usually implemented by executable assertions in software. Error detection codes and duplexing and comparison are based on redundancy in the information representation, either by adding control bits to the data (checksum), or characterizing the data in a new form accommodating the redundancy.

1.2 Formal Methods and Fault-Tolerance

In the last few decades, significant effort has been made to use formal methods to specify and verify fault-tolerant systems. In this section we review briefly some of these approaches.

Program Verification: There are many case studies in which formal methods have been used to verify correctness of distributed and/or real-time protocols [Kulkarni et al., 1999; Schlichting and Schneider, 1983] in the dependability area. Moreover, an important point to achieve an amenable verification is to decompose the fault-tolerant program into several components. In [Cristian, 1985], the author introduced a framework for the design and verification of the correctness of fault-tolerant programs. In this approach, faults are modeling as operations performed at random time intervals by the system's adverse environment. He investigates programs that are subject to hardware faults and processor crashes.

The core of his work is the extension of Flloyd/Hoare logic with rules which enable reasoning about crashes and failures in hardware devices. Schlichting and Schneider [Sinha and Suri, 1999] propose a formal methodology to specify and verify dependable computing systems. Their approach rests on the notion of *fail-stop processors*, i.e., processors that automatically halt in response to any internal failure, with the intention of avoiding the failure becoming visible by the user. An axiomatic verification technique is described to verify programs running on these kinds of processors. In [Arora, 1992; Arora and Gouda, 1993], the authors characterize fault-tolerant programs by means of predicates. In order to distinguish those states which are free of errors, they use invariants. Moreover, their framework is based on the concepts of *closure* and *convergence* used to define fault-tolerance features. The former is the property of a system of remaining in a certain set of "legal" states during the occurrence of faults. The latter is the case when faults stop occurring, and the program eventually reaches a state where the invariant describing the correct states is satisfied. They demonstrate the applicability of their definitions for specifying and verifying the fault-tolerance properties of a variety of digital and computer systems (e.g., Atomic Commitment Protocol, Data Transfer Protocol, and a Delay-Insensitive Circuit).

Program Transformation: In general, in this approach, the idea is to transform a fault-intolerant program into a fault-tolerant program by adding necessary fault detection and correction components. Early work on transformation for fault-tolerance has focused on recovery mechanisms. For instance, in [Liu and Joseph, 1992, 1993; Peled and Joseph, 1994], they used the method of checkpointing with forward and backward recovery. Arora and Kulkarni in [Arora and Kulkarni, 1998b] introduced the idea of detector and corrector components, in which a fault-intolerant program is transformed to a fault-tolerant one by adding these components. The authors have proven that these components are necessary and sufficient to achieve the usual types of fault-tolerance. In more recent works, Kulkarni et al. [Kulkarni and Arora, 2000; Kulkarni and Ebnenasir, 2003, 2004; Ebnenasir et al., 2008; Bonakdarpour et al., 2012], have been working on adding fault-tolerance concerns to existing programs under the occurrence of faults. The reader can find an interesting survey of transformational

approaches of fault-tolerant systems provided by Gärtner in [Gärtner, 1999b].

Self-Stabilizing Programs: provide a built-in safeguard versus transient failures that might corrupt the data in a distributed system. The concept was introduced by Dijkstra in [Dijkstra, 1974] and Lamport in [Lamport, 1985] showed its relevance to fault-tolerance in distributed systems. However, serious work only began in the late nineteen-eighties. An interesting survey of self-stabilizing algorithms can be found in [Schneider, 1993]. In this context, faults are transient (these occur once and then disappear), which means that it is not necessary to have as an assumption a bound on the number of failures. An elemental idea of self-stabilizing algorithms is that the distributed system may be started from an arbitrary global state. Then the system eventually reaches a correct global state, named a *legitimate* or *stable* state. An algorithm is self-stabilizing if the following two conditions are satisfied. First, from any initial illegitimate state it reaches a legitimate state after a finite number of node moves. Second, from any legitimate state the next state is a legitimate state. In case that the system is affected by other classes of faults like Byzantine, a self-stabilizing system can not guarantee that the system is able to operate correctly. Several other works have proposed a formal framework to reason about self-stabilization, some of them are: [Katz and Perry, 1993; Prasetya and Swierstra, 2005; Lentfert and Swierstra, 1993], mainly extending the logic of UNITY.

Theorem Provers: Many significant problems have been, and continue to be, solved using Automated Theorem Proving. The fields where the most notable successes have been achieved are mathematics, software creation and verification, and hardware verification. In recent years, there have been many examples of significant applications in the area of safety-critical systems. For instance, PVS [Owre et al., 1992] is a (semi-)automatic theorem prover that has been successfully used in various applications, including diagnosis and scheduling algorithms for fault-tolerant architectures, and requirements specification for portions of the space shuttle flight control system. Some interesting examples are the verification of the AAMP5 avionics processor [Srivas and Miller, 1996] and the formal verification for fault-tolerant time-triggered algorithms [Rushby, 1999]. In other works, a formal verification system based on the use of automated reasoning techniques is described to validate fault-tolerance in [Jr. et al., 1989].

Moreover, Hickey in [Hickey, 1999] presents an architecture, implemented in the MetaPRL logical framework, for distributing tactic proving over large groups of processors using the Ensemble group communication system. Other theorem provers have been used to prove fault-tolerant properties in specific scenarios; examples can be found in [Moreau, 2006; Barsotti et al., 2007; Zhang, 2008; Mantel and Gärtner, 2000; Lincoln and Rushby, 1993; Qadeer and Shankar, 1998].

Model Checking: On the other hand, many researchers have been used model checking techniques in order to verify and validate fault-tolerant systems. For example, in [Schneider et al., 1998] the requirements of an embedded spacecraft controller were validated using SPIN [Holzmann, 1997], and in [Gnesi et al., 2000] several properties of a railway control system were proven using SPIN. Moreover, in [John et al., 2013b,a], the authors focus on model checking of fault-tolerant distributed algorithms like Paxos. They show how one can model this basic fault-tolerant distributed algorithm in Promela such that safety and liveness properties can be efficiently verified in SPIN. A more general approach is taken in [Yokogawa et al., 2001], where programs written in the programming language introduced in [Arora, 1992] are translated to SMV [McMillan, 1992] and then properties of a given program are verified using the SMV tool. In [Bruns and Sutherland, 1997], the authors present an approach to the model checking of fault-tolerant systems based on process algebra (CCS-based). They define new, special-purpose process operators to model faults and fault-handling mechanisms. They argue that these definition has technical advantages where both the size and generality of model checking can be reduced. Ezekiel and Lomuscio [Ezekiel and Lomuscio, 2009] combine automatic fault injection with model checking to verify fault-tolerance in multi-agent systems (MAS). They present a generic method to mutate a model of a correct system behavior into a faulty one, and discuss how the mutated model can be used to reason about fault-tolerance, which includes recovery behavior from faults. They demonstrated the application of their work by injecting automatically faults into a sender-receiver protocol, and verifying temporal and epistemic specifications of the protocols' fault-tolerance using the MCMAS model checker [Lomuscio and Raimondi, 2006]. Some other interesting work can be found in [Bernardeschi

et al., 2002; Gnesi et al., 2005].

Process Algebra based Approaches: In several works, notions coming from process algebra [Milner, 1980] are used to specify and verify fault-tolerant concurrent programs. For example, Peleska in [Peleska, 1991] models fault-tolerance achieved by dynamic redundancy in CSP, in which he proposes a general approach for proving correctness properties. Amadio and Prasad [Amadio and Prasad, 1994] present an extension to the π-calculus with locations and failures, and gives example of small fault-tolerant programs. Krishnan [Krishnan, 1994] proposed a CCS-based approach in which he models majority voting, using pre-orders to characterise relativised fault-tolerance, and the notion of fault injection. Riely and Hennessy [Riely and Hennessy, 1997] use process algebra to describe a model of locations and failures providing a number of semantic equivalences. Janowski in [Janowski, 1995, 1997], investigated various notions of bisimulation with the aim of capturing fault-tolerant properties, in a CCS-based approach. In addition, these approaches have been applied to several case studies. For instance, Jifeng and Hoare [He and Hoare, 1987] uses CSP to describe and prove correct a distributed recovery algorithm. Bruns in [Bruns, 1992] models railway interlocking using CCS, including failure behaviors and a failure-handling mechanism, verifying safety properties. Gilmore et al. [Gilmore et al., 1995] uses a stochastic process algebra to model performance of robot control with and without failures. Finally, Bernardeschi et al. [Bernardeschi et al., 1998] uses process algebra to verify correctness properties of the GUARDS project, in which he represents faults as actions, and uses a standard concurrency tool kit.

Specification Languages: Several formal languages and frameworks have been used to formalize and to prove properties of specific examples of fault-tolerant systems. In general, these do not have any special construct for modelling fault-tolerant systems in terms of differences between correct, expected or ideal behaviour and incorrect, unexpected or abnormal behaviour. Hence, these features are encoded using ad-hoc mechanisms as part of the general design. For example, one well-known case study is the Byzantine generals problem formalized by Lamport and Merz in [Lamport and Merz, 1994] using TLA+ [Lamport, 1994]. Another popular area of research in the community of fault-tolerant systems

is Railway Signalling [Abdelouahab and Braga, 2008; Abrial, 2006; Guiho and Hennebert, 1990]. This demands the use of an ultra reliable fault-tolerant system since it is directly related to the movement of passenger trains and a fault in the system may cause a train collision and the loss of human life. In [Abrial and Hallerstede, 2007], the Event-B language [Abrial, 2006] is used to model a software controller responsible for the management of the movements of trains on a track network. The authors in this work focused on the prevention of errors and also on their tolerance. Moreover, the formal modelling of this case study included not only the software model but also a detailed model of its environment. Another example using Event-B is given in [Yadav and Butler, 2009], where a broadcast protocol is specified and verified. In [Kang and Jackson, 2008], a file system is specified with the lightweight modeling language Alloy [Jackson, 2006], and verified using the Alloy analyzer. Duration calculus [Chaochen et al., 1991] has been designed for reasoning about real time systems; several examples related to fault-tolerance and real time systems are described in [Chaochen and Hansen, 2004] (e.g., a gas burner).

1.3 Deontic Formalisms and Fault-Tolerance

Deontic logic is a variation of modal logic whose purpose is to logically capture the notion of *norm*. Deontic formalisms have been used in computer science for different purposes such as database security, reactive system specification, artificial intelligence and legal reasoning; see [Wieringa and Meyer, 1993] to read a more detailed survey. Regarding fault-tolerance, the main motivation for using deontic logic that many researchers put forward is that norms and normative reasoning arise naturally in fault-tolerance and it appears inviting to include deontic predicates into existing formal languages. Consequently, this allow us to differentiate between normal and abnormal behaviour. Other researchers (e.g., [Wieringa and Meyer, 1993]) argue that deontic logics are appropriate for reasoning about fault-tolerant systems due to the similarity between fault occurrences in computer systems and the situation in legal and normative systems where violations of laws or regulations occur. We present the following works which use deontic logics for reasoning about problems related to fault-tolerance.

[Carmo and Jones, 1996a] present an extension of standard deontic logic for database specification. They investigate how to deal with problems when a violation arises because of a norm violation, focussing on how to react to, recover from, or repair this violation using their logic. Moreover, the authors consider distinct kinds of integrity constrains and a distinction is made among hard (necessary) and soft (deontic) constraints. The soft integrity constraints admit violations; therefore, the notion of recovery (from the violation of static or state constraints) is characterized. The authors do not cover the concept of transition constraint, in which permitted changes can be made on database states, thus only norms concerning states are studied.

In [Lomuscio and Sergot, 2004], three variations of the bit-transmission problem are formalized by means of a *deontic interpreted logic*. In this approach, a deontic machinery is developed based on the classification of agents' states into "green" and "red". These terms correspond to the correct and incorrect functioning behaviour respectively. We use this idea in our work, see Chapter 2 for more details on our framework. Finally, Lomuscio and Sergot state that the extension incorporating colored transitions in their framework is left for future work.

[Khosla and Maibaum, 1987] present a deontic logic for the specification of systems, but fault-tolerance concerns are not discussed in this work. One of the important points made by the authors is that this logic can be used for the *prescription* and *description* of computing systems and this can also be used for the characterization of abnormal executions. The *description* of a system action is usually given in terms of its precondition and postcondition (what the system does). On the other hand, the *prescription* of a system is understood as what the system should do, stated using deontic predicates.

Fiadeiro and Maibaum in [Fiadeiro and Maibaum, 1991] present an approach in which from a deontic specification of a system, they show how to reason about safety and liveness properties of the normative behaviors of that system. In this approach, they can distinguish between normal and abnormal behaviors, allowing us to prove the properties that hold in normal situations and those that are fulfilled in consequence of an unexpected behaviour.

In [Kent et al., 1993], the authors present a deontic action logic which is used to formalize a library system. Through this example, they explain how to specify temporal constraints and error recovery. However, the logic is sketched and only a

partial axiomatization is presented.

In a more recent work by Castro and Maibaum [Castro, 2009; Castro and Maibaum, 2007, 2009a,c], they introduce a new deontic action logic for specification and analysis of fault-tolerant systems. The main idea behind this mathematical framework is to use axiomatic theories to specify systems. They provide two different deductive systems; the first one is a standard (Hilbert style) deductive system, and the second one is a tableaux system, which can be applied automatically to prove properties of specifications. Several case studies are provided, such as the Diarrheic Philosophers, the Muller C-element, a Simple Train System, Processor Coolers, etc. In [Castro and Maibaum, 2010], the authors extended their logic to a first-order deontic action version which has standard quantifiers of first-order logic and algebraic operators for actions similar to those of the propositional deontic logic. Finally, the authors in [Castro et al., 2011] present a branching time temporal logic for fault-tolerant system verification called dCTL, which will provide the foundation of our work, see Subsection 2.3.1 for more details.

1.4 Automated Synthesis

Automated synthesis of programs is an algorithmic approach where a program is constructed starting from a set of properties. It was originally suggested by Church [Church, 1963] and subsequently solved by two techniques [Büchi and Landweber, 1969; Rabin, 1972]. This problem has been extensively studied for some time through different approaches [Alur et al., 1996; Attie and Emerson, 2001; Emerson and Clarke, 1982; Lafortune and Lin, 1991; Lin and Wonham, 1990; Maler et al., 2006; Manna and Wolper, 1984; Thomas, 2002; Wallmeier et al., 2003], in which the synthesis problem has been analyzed from two points of view: *synthesis from a specification* and *synthesis via solving games*. The first one is based on the decision procedure that verifies the satisfiability problem of the corresponding specification language, whereas the second one is based on the the realizability problem [Abadi et al., 1989] of the corresponding specification language. Moreover, one important difference between these methods is that the former only considers synthesis of closed systems, and the latter considers open systems, where the system interacts with the environment.

In the following Subsections 1.4.1 and 1.4.2, we discuss the line of research on

synthesis of closed and open systems, respectively. Finally, we review in Subsection 1.4.3 the most important works in the area of automatic synthesis of fault-tolerant systems.

1.4.1 Program Synthesis in Closed Systems

In general, synthesis methods for closed systems concentrates on deriving the synchronization skeleton of a program from a given specification written in a specific temporal logic, e.g., [Emerson and Clarke, 1982; Manna and Wolper, 1984; Attie et al., 2004; Attie, 1999; Attie and Emerson, 2001]. These methods are all based on the satisfiability problem for the input specification language. One drawback on these techniques is that any change in the input specification requires the synthesis process to be restarted from the beginning.

The seminal work in this area is due to Emerson and Clarke [Emerson and Clarke, 1982], who propose a tableau-based method for deriving a finite state model from CTL specifications. Essentially, their synthesis method builds a tableau containing all potential models. Hence, if a formula f is satisfiable, then a model of f exists in the tableau. As a output of their synthesis method, a synchronization skeleton of the model is extracted from the tableau. In short, a tableau for a CTL formula is a finite directed AND/OR graph. The construction process of this graph involves the following two steps: (1) setting the formula as the root of the tableau, and (2) expansion of frontier AND-nodes and OR-nodes until the set of frontier nodes is empty (i.e., a node without successor). Finally, pruning rules are applied in the tableau to delete inconsistence nodes. If the root of the tableau is not removed, then the input formula f is satisfiable.

Similarly, Manna and Wolper [Manna and Wolper, 1984] propose a method for synthesizing distributed communicating processes from a formula written in Propositional Temporal Logic (PTL) using a tableau-based satisfiability algorithm for PTL. Their synthesis method produces two outcomes: either the input specification is unsatisfiable; or, it produces a model graph from which all possible models of the input specification can be extracted. The final model can be viewed as the synchronizer process and the other inter-processes can be obtained as restrictions of that model.

Attie and Emerson investigate the problem of synthesizing concurrent programs from CTL specifications in a series of papers. For example, they extend in [Attie and

Emerson, 1998] the synthesis methods introduced in [Emerson and Clarke, 1982] and [Manna and Wolper, 1984] based on the behavioral similarity of processes and thus avoiding the exponential overhead as a result of the state explosion problem. In a different work, the authors address the problem of synthesizing distributed processes following the synthesis method in [Emerson and Clarke, 1982], where they included atomic read/write actions [Attie and Emerson, 2001].

1.4.2 Program Synthesis in Open Systems

In reactive systems, a program interacts with the environment, where the ideal behavior of a correct program should satisfy its specification considering all environments. Game-theoretic approaches consider the situation as a game between the environment and the program. A correct program can be then viewed as a winning strategy in this game. It is well-known that satisfiability of the specification is not sufficient to guarantee the existence of such a strategy. In [Abadi et al., 1989], the authors called specifications for which a winning strategy exists *realizable*. Game theory and control theory are closely related to the problem of program synthesis. We analyze briefly both approaches.

Synthesizing controllers in control theory: The general scenario is that there is a system, called a *plant* in this context, which interacts with an environment. The main goal is to design a *controller* (modeled as a discrete-event system (DES)) which will interact with the system, observing and controlling it using its own inputs to it, in order to make the system behave in an appropriate manner. The seminal work in the area of controller synthesis is due to Ramadge and Wonham [Ramadge and Wonham, 1987, 1989]. In this work, the problem was investigated in an automata-theoretic framework and was demonstrated to have reasonable sub-classes where the problem is decidable. Research in the community of supervisory control of DES has been studying different issues, such as partial observability (where the controller has a restricted capability of observing the plant) [Lin and Wonham, 1990; Rudie and Wonham, 1992; Thistle and Lamouchi, 2009], supremal controller (the controllers that present the least restriction of the system) [Kumar et al., 1991], decentralized control (several controllers are involved, each one having access and control of one

part of the system) [Kumar and Shayman, 1997; Wong and Wonham, 1998]. Other interesting works in the area can be explored in [Kupferman et al., 2000; Madhusudan and Thiagarajan, 1998, 2001; Maler et al., 1995].

Synthesizing strategies for two-player games: In general, game theoretic approaches for the synthesis of controllers and reactive programs [Pnueli and Rosner, 1989a] are based on the model of two-player games [Thomas, 1995]. In such games, a program and its environment are the players. They interact through a set of interface variables, where the environment is restricted to update only these interface variables. Game theoretic methods are based on the theory of tree automata [Thomas, 1990]. The specification of a system is represented by an automata, then a synthesis algorithm tests whether there exists a tree acceptable by the tree automata, this is better known as the nonemptiness problem of tree automata. If the synthesis method returns as a result that the language of the tree automata is nonempty, then the specification is named *realizable*. Hence, a model of the synthesized program exists. In [Pnueli and Rosner, 1989a], Pnueli and Rosner investigate the problem of synthesizing synchronous open reactive modules from a specification given in Linear Temporal Logic (LTL). In a subsequent work [Pnueli and Rosner, 1989b], they generalized their method by means of a technique for synthesizing asynchronous reactive modules. One drawback of this approach is the high complexity of the synthesis process. However, other researchers [Alur and La Torre, 2004; Harding et al., 2005; Pnueli et al., 1998; Wallmeier et al., 2003] have obtained some interesting results, where the synthesis problem can be solved in polynomial time, by restricting the specification of the design to be synthesized to simpler automata or partial fragments of LTL. Particularly, in [Bloem et al., 2012] the authors present an algorithm that solves realizability and synthesis for a subset of LTL, which can essentially be viewed as a generalization of the results of [Pnueli et al., 1998] and [Alur and La Torre, 2004]. An interesting result is that the synthesis algorithm works in cubic time. The authors have shown that the approach can be applied to a wide class of formulas, which covers the full set of *generalized reactivity (1)* properties (GR(1)). Some other interesting related works are [D'Ippolito et al., 2010, 2011]. In more recent work, Wallmeier, Hütten, and Thomas [Wallmeier et al., 2003] introduce a synthesis algorithm

for finite state controllers by solving infinite games over finite state spaces. The authors model the winning constraint of the game graph by safety conditions and a set of request-response properties as liveness conditions, and transforming this game into a Büchi game.

1.4.3 Automated Synthesis of Fault-Tolerance

The problem of synthesizing fault-tolerant systems has been studied in the literature from different perspectives. The primary work in this area is due to Attie, Arora, and Emerson [Attie et al., 2004], where they synthesize fault-tolerant concurrent programs from CTL specifications, based on the tableau-based method defined by Emerson and Clarke in [Emerson and Clarke, 1982]. They require as input of their synthesis algorithm: (1) a problem specification expressed in CTL, (2) a fault specification involving a set of auxiliary atomic propositions and a set of fault actions which represent faults expressed as guarded commands over the atomic propositions, (3) the coupling specification, which is also a CTL formula relating the atomic propositions in the problem specification with those in the fault specification, and (4) a level of fault-tolerance. Initially, the synthesis algorithm builds a tableau based on the tableau-based method defined in [Emerson and Clarke, 1982], which contains normal nodes and transitions representing the intended behaviour of the program in the absence of faults. Then, the current tableau is augmented with fault nodes and transitions by applying the fault actions introduced in the fault specification to every state of the generated model. This step introduces those states reached under the occurrence of faults. Subsequently, recovery transitions are generated in order to produce a recovery behavior where the desired user's level of fault-tolerance (e.g., masking, nonmasking, or failsafe) is satisfied. Finally, a set of deletion rules is applied to remove all nodes that are either propositionally inconsistent, or do not have enough successors, or are labeled with a CTL eventuality formula which is not fulfilled. If the root of the generated tableau is not removed, then a final model is embedded in the final tableau. In the last step of the algorithm, an unravelling process is applied to extract the fault-tolerant program from the generated tableau. We revisit this work in more detail in Chapter 4.

A different approach for automatic addition of fault-tolerance is presented by Kulkarni et al. [Kulkarni and Arora, 2000; Kulkarni and Ebnenasir, 2003, 2004;

Ebnenasir et al., 2008; Bonakdarpour et al., 2012], where the main idea is to add fault-tolerance concerns to existing programs under the occurrence of faults. In more detail, the synthesis method consists of transforming an existing fault-intolerant program into a fault-tolerant version. In particular, they introduce sound and complete algorithms for adding the three levels of fault-tolerance, masking, nonmasking, and failsafe, to programs. They require as input of these algorithms a fault-intolerant program, a safety specification, and a set of fault transitions. The output of the algorithms is a fault-tolerant program satisfying a desired level of fault-tolerance in the presence of faults, and the safety and liveness properties of the fault-intolerant program are fulfilled in the absence of faults.

The initial work by Kulkarni and Arora [Kulkarni and Arora, 2000] introduced synthesis techniques for automated addition of fault-tolerance specifically for centralized and distributed programs. Specially, for centralized programs they introduce sound and complete algorithms for adding the different levels of fault-tolerance (masking, nonmasking, and failsafe) in polynomial time. Regarding distributed programs, the authors show that the problem of adding masking fault-tolerance to distributed programs is NP-complete in the size of the input program's state space. In order to deal with this complexity, the authors in [Kulkarni et al., 2001] define a set of heuristics to solve the problem of synthesizing distributed masking programs in polynomial time.

Kulkarni and Ebnenasir in [Kulkarni and Ebnenasir, 2004] studied the problem of synthesis of multitolerant programs, i.e., those programs that tolerate multiple classes of faults, in which different fault classes may require distinct levels of fault-tolerance. The novelty of their synthesis approach is that the addition of fault-tolerance is performed in a stepwise fashion. They obtained interesting results depending on the diverse combination of fault classes. For example, if it is intended to add either failsafe or nonmasking fault-tolerance considering one class of faults and masking fault-tolerance to a different class of faults, then this addition has been proved by the authors to take polynomial time with respect to the size of the state space of the fault-intolerant program. However, if it is the case that one intends to add failsafe fault-tolerance with respect to one class of faults and nonmasking fault-tolerance considering another class of faults, then the problem result in being NP-complete.

Bonakdarpour, Kulkarni, and Abujarad in [Bonakdarpour et al., 2012] concentrate on automated addition of masking fault-tolerance to fault-intolerant distributed

programs. The main contribution of this work is the development of an efficient symbolic heuristic based on Ordered Binary Decision Diagrams (OBDDs), which are used to represent programs, faults, specifications, etc. They showed that in spite of the synthesis problem being NP-complete in the size of the input program's state space, synthesizing masking distributed programs from a fault-intolerant version is feasible in practice. They validated their work with different case studies like the Byzantine agreement problem, token ring, Infuse, etc. Another efficient method is presented by Ebnenasir [Ebnenasir, 2007], where the author develops a divide-and-conquer method for automatic addition of failsafe fault-tolerance to fault-intolerant distributed programs. The efficiency of this approach is based on the utilization of processing power on separate machines in a parallel platform.

In the context of controller synthesis, Chao and Lim [Cho and Lim, 1998] introduced in this area the idea of transforming a fault-intolerant system into a fault-tolerant one. Moreover, Girault and Rutten [Girault and Rutten, 2009] present a framework for automating the addition of fault-tolerance using Discrete Control Synthesis (DCS). They use labeled transition systems (LTS) to specify the several concurrent parts of the system, where they model different kinds of faults (e.g., processor crash, Byzantine faults, value corruption) by uncontrollable actions in a LTS. Then, given a fault-intolerant program, DCS is used to synthesize a fault-tolerant version of the input program satisfying the fault-tolerance requirements under the specified fault hypothesis.

Other interesting works in the area of synthesis of fault-tolerance can be explored in [Kulkarni and Ebnenasir, 2005; Gärtner and Jhumka, 2004; Marchand and Samaan, 2000]. In Section 6.1 of Chapter 6 we compare our synthesis approach with some of these works described above.

1.5 Aim of the thesis

The specific goal of this dissertation is the development of a synthesis method for fault-tolerant programs. More specifically, we are interested in studying the problem of automatically synthesizing fault-tolerant systems from logical specifications, i.e., the problem of automatically constructing a fault-tolerant component implementation

from a logical specification of the component, and the system's required level of fault-tolerance.

Several algorithms have been presented in the literature for synthesis of reactive system from their temporal logic specification (see Section 1.4). With regards to automated synthesis of fault-tolerance from a specification, the primary work is due to Attie, Arora, and Emerson [Attie et al., 2004], where they present an algorithm for synthesizing fault-tolerant programs from CTL specifications, based on the tableau-based method defined by Emerson and Clarke in [Emerson and Clarke, 1982]. Most of these works consider CTL and CTL* as the temporal logic specification languages for the input of their synthesis methods. It is well-known that these logics have important applications in model checking [Clarke et al., 2001]; however, these logics are not specialized for describing properties of fault-tolerant systems. For this reason, we hypothesize that it is important to provide to the users, from the beginning, a more natural way for specifying properties of fault-tolerant systems. In our work, we support the idea that deontic logics are appropriate to reason about fault-tolerant systems because they allow us to distinguish between normal and abnormal situations; by using this, we can characterize what may be wrong and what to do about it. Moreover, benefits of deontic logics include: they have a language to express normative reasoning (permission, obligation, forbidden) and it is easy to mix them with temporal logics, and therefore to gain the good properties of temporal frameworks to verify systems. We present several case studies explaining how we use deontic operators in order to specify fault-tolerant programs.

Another important issue in our research is how faults are injected during the synthesis process in order to produce a program which tolerates those faults, satisfying one of the levels of fault-tolerance. In our investigation, we observed that in many approaches (e.g. [Attie et al., 2004; Kulkarni and Arora, 2000; Kulkarni and Ebnenasir, 2004; Ebnenasir et al., 2008]) faults are given explicitly as part of the behavior model of the system. This means that the user has to provide the faults as input for these synthesis methods in order to obtain a fault-tolerant program that tolerates those faults. We observe that these faults should be known in advance by the users. Similarly to the above approaches, we also allow users to lists the possible faults that can affect the system. Additionally, we investigate how we can inject faults in our framework, but trying to do it from the obligations stated using the deontic operators

in the input specification.

Moreover, another question that requires attention is the problem of representing the different fault-tolerance properties (masking, nonmasking, and failsafe) desired for the synthesized program. These play an important role in the synthesis algorithm when we have to check if the chosen level of fault-tolerance required by the user is fulfilled. In our case, we propose an alternative formal approach for dealing with the analysis of fault-tolerance, which allows for a fully automated analysis, and appropriately distinguishes faulty behaviors from normal ones. This approach provides a formalism for modeling fault-tolerant systems that features a built-in notion of abnormal transition, to capture faults. In this setting, fault-tolerance is characterized by defining simulation/bisimulation relations, between the desired "fault-free" or "ideal" program, and that which tolerates or deals in some way or another with faults. We present the foundations of this approach for representing the fault-tolerance properties and also the different algorithms for fully automated analysis of the three levels of fault-tolerance.

Finally, we are concerned about how our approach works in practice, so we present several case studies showing how to use deontic logics and also we present a software tool which is used to provide experimental results about our synthesis method.

1.6 Outline of the dissertation

In Chapter 2, we review briefly the notions needed to tackle the rest of the thesis. We start by reproducing the basic definitions of Kripke and colored Kripke structures; then we take a look at Computation Tree Logic (CTL). We continue with an overview of deontic logics and we present dCTL-: a fragment of a branching time temporal logic with deontic operators, especially designed for fault-tolerant component specification. Finally, we describe our model of computation.

In Chapter 3, we introduce the first core of the thesis: a formal characterization of fault-tolerant behaviors (masking, nonmasking, and failsafe) of computing systems via simulation relations. In addition, we present the corresponding algorithms for checking automatically fault-tolerance properties in polynomial time, i.e., to verify that a system behaves in an acceptable way even subject to the occurrence of

faults. Finally, we demonstrate the practical application of our formalization through some well-known case studies, which illustrate that the main ideas behind most fault-tolerance mechanisms are naturally captured in our setting.

Chapter 4 addresses the second core of the dissertation: our synthesis method. We present the extension of a synthesis algorithm for CTL to cope with dCTL- specifications. Moreover, we explain the details of each of the algorithms for the three degrees of fault-tolerance, as well as their complexity. Finally, we prove some properties like soundness and completeness of our method.

In Chapter 5 we present several case studies to show how the logical system presented in Chapters 3 and 4 can be used in practice. Moreover, we present our tool syntdctl which was used to synthesize some of these examples.

Finally, in Chapter 6, we conclude. We present the related work in the literature of automated synthesis of fault-tolerant programs. Additionally, we present a detailed road map for future work, and make concluding remarks.

Chapter 2

Preliminary Concepts

In this chapter, we formally define the fundamental elements of our framework. We will start by reproducing the definition of standard Kripke structures, colored Kripke structures, and we continue giving a brief overview of Temporal Logics: Linear Time-Temporal Logic and Branching Time Logic. Moreover, we present a variation of a branching time temporal logic with deontic operators called dCTL, which will provide the foundation of our work. Finally, we describe our model of computation.

2.1 Kripke Structure

Kripke structures [Kripke, 1963] are traditionally used to interpret modal or temporal logic formulas as well as for characterizing the dynamic behavior of reactive systems [Clarke et al., 2001].

Definition 2.1.1. (Kripke Structure) Let AP be a set of atomic propositions. A Kripke structure over AP is a 4-tuple $\langle S, I, R, L\rangle$, where S is a set of elements called *states*, $I \subseteq S$ is a set of *initial states*, $R \subseteq S \times S$ is a *transition* relation between states, and $L : S \to 2^{AP}$ is an interpretation function, which denotes the set of atomic propositions that are true in each state.

Given a Kripke structure $M = \langle S, I, R, L\rangle$, the interpretation of logical connectives and modal operators in a modal logic can commonly be defined by resorting to L and the structure of R. In the case of temporal logics, it is generally necessary to utilize the notion of *trace* to define the semantics of some operators.

2.1.1 Colored Kripke Structures

Colored Kripke structures are a simple variation of Kripke structures that we will use for describing fault-tolerant systems. We reproduce its definition as introduced in [Castro et al., 2011]:

Definition 2.1.2. (Colored Kripke Structure) Given a set of propositional letters $AP = \{p, q, s, \dots\}$, a *colored Kripke structure* is a 5-tuple $\langle S, I, R, L, \mathcal{N} \rangle$, where S is a set of states, $I \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation, $L : S \to 2^{AP}$ is a labeling function indicating which propositions are true in each state, and $\mathcal{N} \subseteq S$ is a set of *normal*, or "green" states. The complement of $\mathcal{N}$ is the set of "red", abnormal or faulty, states. Arcs leading to abnormal states (i.e., states not in $\mathcal{N}$) can be thought of as faulty transitions, or simply *faults*.

Given a colored Kripke structure $M = \langle S, I, R, L, \mathcal{N} \rangle$, a *trace* is a maximal sequence of states, whose consecutive pairs are in R. That is, a sequence:

$$s_0 s_1 s_2 s_3 \dots$$

is said to be a trace of M when $s_i \in S$ and $s_i R s_{i+1}$ for every i. Note that traces may be infinite or finite. When a trace of M starts in an initial state, it is called an *execution* of M, and the set of executions of a structure M is denoted by $\mathcal{TR}(M)$. Normal executions are those transiting only through green states; the set of normal executions is denoted by $\mathcal{NT}(M)$. We assume that, in every colored Kripke structure, for every normal state there exists at least one successor state that is also normal, and that at least one initial state is green. This guarantees that every system has at least one normal execution, i.e., $\mathcal{NT}(M) \neq \emptyset$, for any M.

Given a trace $\sigma = s_0 s_1 s_2 s_3 \dots$, the ith state of σ is denoted by $\sigma[i]$, and the final segment of σ starting in position i is denoted by $\sigma[i..]$. Moreover, we distinguish among the different kinds of outgoing transitions from a state. We denote by $\dashrightarrow$ the restriction of R to faulty transitions, and $\to$ the restriction of R to non-faulty transitions. We define $Post_N(s) = \{s' \in \mathcal{N} |\ s \to s'\}$ as the set of successors of s reachable via non-faulty (or good) transitions; similarly, $Post_F(s) = \{s' \in S |\ s \dashrightarrow s'\}$ represents the set of successors of s reachable via faulty arcs. Analogously, we define $Pre_N(s')$ and $Pre_F(s')$ as the set of predecessors of s' via normal and faulty transitions, respectively. Moreover, $Post^*(s)$ denotes the states which are reachable

from s. Without loss of generality, we assume that every state has a successor; this is a standard assumption in temporal logics [Baier and Katoen, 2008]. We denote by $\Rightarrow^*$ the transitive closure of $\dashrightarrow \cup \rightarrow$.

2.2 Temporal Logics

In [Pnueli, 1977], Pnueli proposed to use temporal logics to specify and verify programs, especially for nonterminating or continuously running concurrent programs (e.g., operating systems). Ever since, temporal logics have been used extensively by computer scientists to build reliable software. Temporal logic extends classical logic by modalities that allow us to referal to the infinite behavior of *reactive* (i.e., concurrent, dynamic, distributed) systems.

Additionally to the operators of classical logic, the fundamental temporal modalities that appear in most temporal logics include the following operators:

X "next" (next moment in time)

F "eventually" (at some future moment)

G "always" (at every future moment)

There are many different temporal logics, where these can usually be classified regarding the underlying nature of time as either *linear time* or *branching time*. On the one hand, in the linear view, time is linear and discrete which means that at any given instant of time there is a single successor. On the other hand, in the branching view, time may split into different courses, where it has a branching, tree-like structure. Branching time models are used when non-determinism is present in specifications. In the following subsections we briefly introduce both approaches.

2.2.1 Linear Time-Temporal Logic

Linear temporal logics (or LTLs for short) were introduced by Pnueli in 1977 in his landmark work [Pnueli, 1977]. This logic is an extension of classical logic including the following temporal operators:

- $\mathsf{X}\ \phi$ (in the next moment in time ϕ is true)

- F ϕ (eventually ϕ is true)
- G ϕ (always in the future ϕ is true)
- $\phi \, \mathcal{U} \, \psi$ (ψ is true at some moment in the future, and *until* ψ becomes true, ϕ is true).

From these temporal modalities, various other operators can be derived such as a weak version of $\phi \, \mathcal{U} \, \psi$ denoted $\mathcal{W}$, i.e., ψ may never be true in the future (see [Manna and Pnueli, 1992]).

The semantics of LTL is given by Kripke structures and traces over it (i.e., paths in the Kripke structure). Paths are usually maximal (though some works consider all the possible paths), and the relation of satisfaction is defined with respect to an instant, a path and a Kripke structure, i.e., $i, \sigma, M \models \phi$ indicates that a formula ϕ is true at the point i of path σ in the structure M. We say that σ *satisfies* a formula ϕ in M, denoted $\sigma, M \models \phi$ iff $0, \sigma, M \models \phi$. The formal semantics for these operators can be found in classic textbooks like [Manna and Pnueli, 1992] and [Emerson and Clarke, 1980].

LTL has been widely used for model checking; SPIN is the most well-known LTL model checker developed by Holzmann [Holzmann, 1997]. Further, LTL model checking using a tableau construction is supported by NuSMV [Cimatti et al., 2000]. The complexity of model checking for LTL was proven to be PSPACE-complete by Sistla and Clarke [Sistla and Clarke, 1985].

2.2.2 Branching Time Logic

Branching time logics consider, for each moment of time, several different possible futures. Each instant of time may hence split into several possible futures. Computation Tree Logic (CTL) is an important branching time temporal logic introduced by Emerson and Clarke [Emerson and Clarke, 1980], which has important applications in the model checking area [Clarke et al., 1986]. This logic allows for the description of properties over Kripke structures, by combining branching operators A ("for all paths or computations") and E ("for some paths or computations") and temporal operators X, F, G, $\mathcal{U}$, where these are immediately preceded by a path quantifier. More precisely, the following are the possible combinations of path quantifiers and temporal operators with their intuitive interpretation:

- A $(\phi \mathcal{U} \psi)$, on all future paths, ϕ is true *until* ψ becomes true.
- AG ϕ, on all future paths, ϕ is always true.
- AF ϕ, on all future paths, ϕ is eventually true.
- AX ϕ, on all future paths ϕ is true at the next moment.
- E$(\phi \mathcal{U} \psi)$, on some future path, ϕ is true *until* ψ becomes true.
- EG ϕ, on some future path, ϕ is always true.
- EF ϕ, on some future path, ϕ is eventually true.
- EX ϕ, on some future path ϕ is true at the next moment.

CTL formulas are interpreted on states over Kripke structures (see [Emerson and Clarke, 1980]).

Regarding the time complexity of the model checking problem for CTL, it results linear in the size of the model and the length of the formula. A very successful CTL model checker called SMV (Symbolic Model Verifier) was implemented by McMillan, which is based on a symbolic OBDD (Ordered Binary Decision Diagrams) based representation of the state space. Moreover, Cimatti et al. developed NuSMV [Cimatti et al., 2000], a variant of SMV.

There has been a lot of discussion about the benefits and drawbacks of using linear and branching temporal logics for reasoning about concurrent programs [Emerson and Halpern, 1986; Lamport, 1980]. More recently, Vardi presented an interesting paper [Vardi, 2001] discussing linear and branching frameworks. One argument in favor of the branching time approach is that the complexity of the model checking problem is polynomial whereas in the linear approach, it is often exponential. An issue that everyone is agreed on is that the expressivenesses of LTL and CTL are incomparable, i.e., there are properties that are possible to express in LTL, but that cannot be expressed in CTL, and vice versa. Extended Computation Tree Logic (CTL*) combines the expressive powers of CTL and LTL offering a more expressive logic. However, the model checking problem for this logic is exponential in the size of the verified formula [Clarke et al., 2001].

These temporal logics allows for the specification of a large number of system properties. In [Manna and Pnueli, 1990], temporal properties are classified by categories where in practice two are the most common: *safety* and *liveness* properties. Informally speaking, the former specifies that "something bad never happens", and the latter specifies that "something good will eventually happen".

2.3 Deontic Logics

Deontic logic is a branch of modal logic, which focuses on the study of the reasoning arising in ethical and moral contexts, which usually involve norms and prescriptions. Usually these logics have two modalities: **P** (permission) and **O** (obligation). After studying the literature, we concluded that there are no standard definitions for these predicates due to the philosophical nature of this logic. Some benefits of deontic logics are: they allow us to distinguish between normal and abnormal situations, they have a rich language to express normative reasoning (obligation, permission, forbidden), they provide a natural level of abstraction in semantic structure (states are divided into "good" and "bad" ones), and finally, it is easy to mix them with temporal logics, and therefore to gain the good properties of temporal frameworks to verify systems.

In [Wieringa and Meyer, 1993] are described many examples of computer science applications, like specification of fault-tolerant systems, the specification of security policies, the automation of contracting, and the specification of normative integrity constraints for databases.

We are interested in deontic concepts in the context of fault-tolerant systems, where some researchers [Carmo and Jones, 1996b; Wieringa and Meyer, 1993; Maibaum and Turski, 1984; Kent et al., 1991; Khosla, 1989] have been studying the application of deontic logics to reason about fault-tolerant systems (see Section 1.3). Particularly, we have studied the work of Castro and Maibaum [Castro, 2009; Castro and Maibaum, 2007, 2009a,c] and a more recent work [Castro et al., 2011]. We focus on the last, which is a branching time temporal logic for fault-tolerant system verification called dCTL. In this section, we present a variation of dCTL, which will provide the foundation of our work.

2.3.1 dCTL: A branching time temporal logic with deontic operators

As we mention above, we model fault-tolerant systems by means of *colored Kripke structures* introduced in the previous subsection 2.1.1.

In order to state properties of systems, we use a fragment of dCTL [Castro et al., 2011], a branching time temporal logic with deontic operators designed for reasoning about fault-tolerant systems. Formulas in this fragment, that we call dCTL-, refer to properties of behaviors of colored Kripke structures, in which a distinction between *normal* and *abnormal* states (and therefore also a distinction between normal and abnormal traces) is made. The logic dCTL is defined over the Computation Tree Logic (CTL), with its novel part being the deontic operators $\mathbf{O}(\psi)$ (obligation) and $\mathbf{P}(\psi)$ (permission), which are applied to a certain kind of path formula ψ. The intention of these operators is to capture the notion of *obligation* and *permission* over traces. Intuitively, these operators have the following meaning:

- $\mathbf{O}(\psi)$: property ψ is obliged in every future state, reachable via non-faulty transitions.
- $\mathbf{P}(\psi)$: there exists a normal execution, i.e., not involving faults, starting from the current state and along which ψ holds.

Obligation and permission will enable us to express intended properties which should hold in *all* normal behaviors and *some* normal behaviors, respectively. These deontic operators have an implicit *temporal* character, since ψ is a path formula. One way of thinking about the semantics of this logic is considering some executions as painted green, these are the correct executions of the system, and others painted red or red and green, and they represent the executions of the system containing faults. The obliged properties are those true in the green executions.

The syntax of dCTL- is defined as follows. Let $AP = \{p_0, p_1, \dots\}$ be a set of atomic propositions. The sets Φ and Ψ of *state formulas* and *path formulas*, respectively, are mutually recursively defined as follows:

$$\begin{aligned} \Phi &::= \top \mid p_i \mid \neg\Phi \mid \Phi \to \Phi \mid \mathsf{A}(\Psi) \mid \mathsf{E}(\Psi) \mid \mathbf{O}(\Psi) \mid \mathbf{P}(\Psi) \\ \Psi &::= \mathsf{X}\Phi \mid \Phi \,\mathcal{U}\, \Phi \mid \Phi \,\mathcal{W}\, \Phi \end{aligned}$$

Other boolean connectives (here, state operators), such as $\wedge$, $\vee$, etc., are defined as usual. Also, traditional temporal operators G and F can be expressed as $\mathsf{G}(\phi) \equiv \phi\ \mathcal{W}\ \bot$, and $\mathsf{F}(\phi) \equiv \top\ \mathcal{U}\ \phi$. We define CTL *liveness formulas* as those CTL formulas that only contain AF and EF temporal operators and $\vee$ and $\wedge$ boolean operators. On the other hand, *safety* formulas are those that only contain AG and EG temporal operators and $\vee$ and $\wedge$ boolean operators. We remark that, the fragment dCTL- differs from plain dCTL by excluding two operators: $\mathbf{R}(\psi)$ (repair or recovery) and $\psi \rightsquigarrow \psi'$, which it represents a conditional between trace properties. Intuitively, $\mathbf{R}(\psi)$ indicates that property ψ holds in every future state, immediately after a fault has occured and $\psi \rightsquigarrow \psi'$ states that, for every normal trace σ starting in the current state, if σ satisfies ψ then it also satisfies ψ', see [Castro et al., 2011] for more details.

Now, we formally state the semantics of the logic. We start by defining the relation $\models$, formalizing the satisfaction of dCTL- state formulas in a state s in a colored Kripke structure M. The definition of $\models$ is as follows:

- $M, s \models \top$
- $M, s \models p_i \Leftrightarrow p_i \in L(s)$, where $p_i \in AP$.
- $M, s \models \neg\varphi \Leftrightarrow$ not $M, s \models \varphi$.
- $M, s \models \varphi \rightarrow \varphi' \Leftrightarrow (M, s \models \neg\varphi)$ or $(M, s \models \varphi')$.
- $M, s \models \mathsf{A}(\psi) \Leftrightarrow$ for every σ such that $\sigma[0] = s$ we have that for every $i \geq 0$ $M, \sigma[i..] \models \psi$.
- $M, s \models \mathsf{E}(\psi) \Leftrightarrow$ for some σ such that $\sigma[0] = s$ we have that for every $i \geq 0$ $M, \sigma[i..] \models \psi$.
- $M, s \models \mathbf{O}(\psi) \Leftrightarrow$ for every $\sigma \in \mathcal{NT}(M)$ such that $\sigma[0] = s$ we have that for every $i \geq 0$ $M, \sigma[i..] \models \psi$.
- $M, s \models \mathbf{P}(\psi) \Leftrightarrow$ for some $\sigma \in \mathcal{NT}(M)$ such that $\sigma[0] = s$ we have that for every $i \geq 0$ $M, \sigma[i..] \models \psi$.

The above satisfaction relation makes use of dCTL- satisfaction for path formulas, whose definition is standard:

- $M, \sigma \models \mathsf{X}\varphi \Leftrightarrow M, \sigma[1] \models \varphi$.
- $M, \sigma \models \varphi \ \mathcal{U} \ \varphi' \Leftrightarrow$ there exists $j \geq 0$ such that $M, \sigma[j] \models \varphi'$ and for every $0 \leq k < j$, it holds that $M, \sigma[j] \models \varphi$.
- $M, \sigma \models \varphi \ \mathcal{W} \ \varphi' \Leftrightarrow$ either there exists $j \geq 0$ such that $M, \sigma[j] \models \varphi'$ and for every $0 \leq k < j$ it holds that $M, \sigma[j] \models \varphi$, or for every $j \geq 0$ we have that $M, \sigma[j] \models \varphi$.

We denote by $M \models \varphi$ the fact that $M, s \models \varphi$ holds for every state s of M, and by $\models \varphi$ the fact that $M \models \varphi$ holds for every colored Kripke structure M.

In order to illustrate the semantics of the deontic operators, let us consider the colored Kripke structure in Figure 2.1, where the set of propositional variables is $\{p, q, r, v, t\}$, and each state is labeled by the set of propositional variables that hold in it. The states that are the target of dashed arcs are abnormal states (those in which something has gone wrong); faulty states are also drawn with dashed lines, while the other ones represent normal configurations. Notice that the unique faulty state in this model is that labeled by t. In this simple model, for every non-faulty execution, $p \wedge q$ is always true. In dCTL- this is expressed by the formula $\mathbf{O}(p \wedge q)$. Note that there also exist normal executions for which $p \wedge q \wedge r$ holds. This fact is expressed as $\mathbf{P}(p \wedge q \wedge r)$. Other deontic operators such as *prohibition* can be expressed by using those introduced above (see [Castro et al., 2011]).

One of the interesting characteristics of dCTL- is the possibility of distinguishing between formulas that state properties of good executions and the standard formulas, which state properties of all possible executions. For every formula φ, a formula φ^N can be built, which captures the same property as φ but restricted to good executions. This leads to the notion of the *normative formula* corresponding to a given formula, and is defined as follows.

Definition 2.3.1. Given a dCTL- formula φ over an alphabet AP, the *normative formula* φ^N corresponding to it is defined by the following rules:

- $(p_i)^N \stackrel{\mathsf{def}}{=} p_i$,
- $(\neg\varphi)^N \stackrel{\mathsf{def}}{=} \neg\varphi^N$,
- $(\varphi \wedge \varphi')^N \stackrel{\mathsf{def}}{=} \varphi^N \wedge \varphi'^N$,

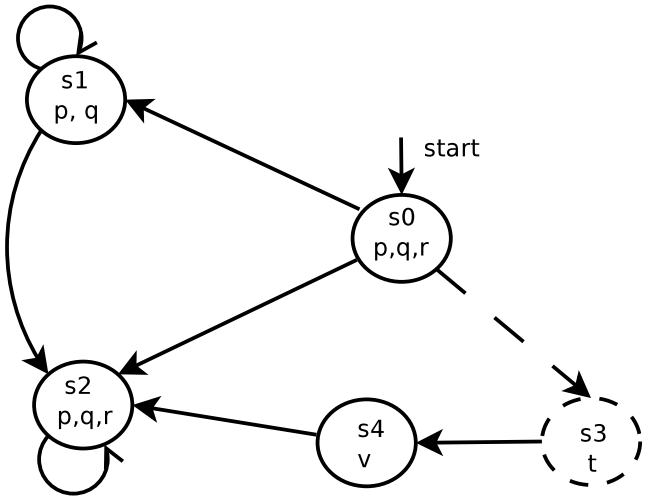

Figure 2.1: A simple colored Kripke structure.

- $(\mathsf{A}\mathsf{X}\varphi)^N \stackrel{\text{def}}{=} \mathbf{O}\mathsf{X}\varphi^N$,
- $(\mathsf{E}\mathsf{X}\varphi)^N \stackrel{\text{def}}{=} \mathbf{P}\mathsf{X}\varphi^N$,
- $(\mathsf{A}(\varphi\,\mathcal{U}\,\varphi'))^N \stackrel{\text{def}}{=} \mathbf{O}(\varphi^N\,\mathcal{U}\,\varphi'^N)$,
- $(\mathsf{A}(\varphi\,\mathcal{W}\,\varphi'))^N \stackrel{\text{def}}{=} \mathbf{O}(\varphi^N\,\mathcal{W}\,\varphi'^N)$,
- $(\mathsf{E}(\varphi\,\mathcal{U}\,\varphi'))^N \stackrel{\text{def}}{=} \mathbf{P}(\varphi^N\,\mathcal{U}\,\varphi'^N)$,
- $(\mathsf{E}(\varphi\,\mathcal{W}\,\varphi'))^N \stackrel{\text{def}}{=} \mathbf{P}(\varphi^N\,\mathcal{W}\,\varphi'^N)$,
- $(\mathbf{O}(\varphi\,\mathcal{U}\,\varphi'))^N \stackrel{\text{def}}{=} \mathbf{O}(\varphi^N\,\mathcal{U}\,\varphi'^N)$,
- $(\mathbf{O}(\varphi\,\mathcal{W}\,\varphi'))^N \stackrel{\text{def}}{=} \mathbf{O}(\varphi^N\,\mathcal{W}\,\varphi'^N)$,
- $(\mathbf{P}(\varphi\,\mathcal{U}\,\varphi'))^N \stackrel{\text{def}}{=} \mathbf{P}(\varphi^N\,\mathcal{U}\,\varphi'^N)$,
- $(\mathbf{P}(\varphi\,\mathcal{W}\,\varphi'))^N \stackrel{\text{def}}{=} \mathbf{P}(\varphi^N\,\mathcal{W}\,\varphi'^N)$.

2.4 Model of Computation

Some remarks are necessary about our model of computation. We take the view of [Arora and Gouda, 1993; Attie et al., 2004; Dijkstra, 1976; Gärtner, 1999a; Kulkarni

Normal Actions:

$$(state = 0) \rightarrow (state := 1)$$
$$(state = 1) \rightarrow (state := 2)$$
$$(state = 2) \rightarrow (state := 0)$$

Faulty Actions:

$$(state = 1) \rightarrow (state := 3)$$
$$(state = 1) \rightarrow (state := 6)$$
$$(state = 3) \rightarrow (state := 4)$$
$$(state = 4) \rightarrow (state := 5)$$
$$(state = 5) \rightarrow (state := 7)$$
$$(state = 6) \rightarrow (state := 7)$$

Figure 2.2: A simple program "Never 7".

and Arora, 2000] and describe programs in a *guarded command* style. A guarded command is composed of a boolean condition over the actual state of the system and an assignment, written as $Guard \rightarrow Command$. These syntactical constructions are called actions, and a program consists of a collection of actions. We can use some actions to represent faults (as done in [Arora and Gouda, 1993; Attie et al., 2004; Gärtner, 1999a]). Furthermore, we can devise distributed systems, where we have several programs interacting concurrently; the interested reader is referred to [Chandy and Misra, 1989] for a detailed introduction to this style of programming. The important point here is that we can map these programs to colored Kripke structures, mapping variable valuations to states and actions to transitions; here green transitions represent non-faulty actions and red ones capture faulty actions. An example of this is shown in Figure 2.3. This is a simple example called *Never 7*, introduced by Bastian Braun in his M.Sc. thesis at University of Mannheim and it was also used by Bonakdarpour in his PhD thesis [Bonakdarpour, 2008], where we have adapted from this last reference. The program has eight states and the system specification requires that state 7 is not reached in the future, and the invariant predicate of the program is the set $\{0, 1, 2\}$. The behavior of this small system can be expressed by the program shown in Figure 2.2. For the sake of simplicity, we assume that we only have boolean variables in our programs; it is straightforward to extend this programming language with other programming types. Note that, in the kinds of programs described above, we may have two actions enabled at the same time; if this happens infinitely often during the execution of a system, we may have

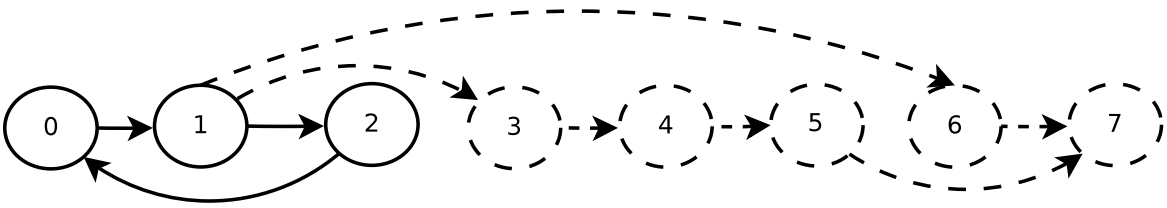

Figure 2.3: Never 7 program.

scenarios where some actions are neglected infinitely often. To avoid such scenarios, we must introduce the notion of *fair executions*. In order to express this fairness assumption, we follow the ideas introduced in [Aminof et al., 2004], where Kripke structures are augmented with fairness conditions. The authors consider a *transition fairness* defined as follow: "*A path π is fair with respect to the transition fairness condition iff all the transitions that are enabled along π infinitely often are also taken along π infinitely often*".

Definition 2.4.1 (Set of Fair Executions)**.** Given a colored Kripke structure $M = \langle S, I, R, L, \mathcal{N} \rangle$ we define the set of fair executions of M as follows:

$$\mathcal{FT}(M) = \{\sigma \mid \sigma \in \mathcal{TR}(M) \text{ and } \forall w, t \in S : \forall i : \exists j > i : t \in Post(\sigma[j]) \wedge \sigma[j] = w \Rightarrow \forall i : \exists j > i : \sigma[j] = t \wedge \sigma[j-1] = w\}$$

We say that a transition $w \rightarrow t$ is enabled in position i in σ, if $\sigma[i] = w$. Definition 2.4.1 says that actions that are enabled infinitely often are executed infinitely often. In practice, fair programs can be implemented by using schedulers, and it is a standard assumption in concurrency. It is worth noting that in our definition of fair executions we also consider faulty actions, that is, the faulty actions that are enabled infinitely often in an execution, will occur infinitely often. We can also introduce fair normative executions which do not take into account faults, as follows:

Definition 2.4.2 (Set of Fair Normative Executions)**.** Given a colored Kripke structure $M = \langle S, I, R, L, \mathcal{N} \rangle$ we define the set of fair normative executions of M as follows:

$$\mathcal{FNT}(M) = \{\sigma \mid \sigma \in \mathcal{TR}(M) \text{ and } \forall w, t \in S : \forall i : \exists j > i : (t \in Post_N(\sigma[j]) \wedge \sigma[j] = w) \Rightarrow \forall i : \exists j > i : \sigma[j] = t \wedge \sigma[j-1] = w\}$$

When useful we denote the set of fair (normative) executions starting in state s by $\mathcal{FT}(M)(s)$ ($\mathcal{FNT}(M)(s)$).

Note that the restriction to fair executions is reasonable when one inspects the way in which faults are distributed in practice. If a fault has a positive probability of occurring (if it has probability 0, it can be deleted from the model), then during an infinite execution it will occur infinitely often. However, in the case that one wants to restrict the number of occurrences of any fault, the program and the specification of the faults can be modified straightforwardly to do so. The restriction of $\models$ to fair (normative) executions is denoted by $\models_f$ ($\models_{nf}$).

Chapter 3

Bisimulation and Fault-Tolerance

Bisimulation and Simulation are rich concepts which appear in various areas of theoretical computer science. Its origins lie in concurrency theory, for instance see Milner [Milner, 1980], in modal logic, see van Benthem for example [van Benthem, 1999], and with respect to temporal logics [Browne et al., 1987] and [De Nicola and Vaandrager, 1995].

In this chapter we present suitable notions of simulation relations that allow us to capture diverse fault-tolerance properties, namely, *masking*, *nonmasking*, and *failsafe* fault-tolerance. In order to define these properties, we follow the basic definitions regarding simulation and bisimulation relations given in [Baier and Katoen, 2008].

We assume that the properties of interest of a system are defined by means of a set of safety and liveness properties (recall that any temporal specification can be written as a conjunction of safety and liveness properties [Alpern and Schneider, 1985]). Basically, in order to check fault-tolerance, we consider two colored Kripke structures for a system, the first one acting as a specification of the intended behavior and the second as the fault-tolerant implementation. A system will be fault-tolerant if it is able to preserve, to some degree, the safety and liveness properties corresponding to its specification, even in the presence of faults. Our main goal is to capture, via appropriate (bi)simulation relations between the system specification and the fault-tolerant implementation, different kinds of fault-tolerance, with different levels of property preservation.

In the following definitions, given a colored Kripke structure with a labeling L, we consider the notion of a sub-labeling: we say that L_0 is a sub-labeling of L (denoted

by $L_0 \subseteq L$), if $L_0(s) = L(s) \cap AP'$, for all states s and some $AP' \subseteq AP$. We also say that L_0 is obtained by restricting AP to AP'. The concept of sub-labeling allows us to focus on certain properties of models.

We remark that the content of this chapter is fully similar to the version submitted for a journal publication [Demasi et al., 2014].

3.1 Masking Fault-Tolerance

Recall that a program is said to be masking fault-tolerant when it continues satisfying part (perhaps all) its safety and liveness specification even under the occurrence of faults. A minor observation about this definition is useful. Usually, when verifying a component, one is interested in the behavior that is observable through its interface; thus, when defining masking fault-tolerance we restrict ourselves to the interface of the component, captured formally by means of the notion of sub-labeling. Let us introduce the notion of masking fault-tolerance simulation.

Definition 3.1.1. (Masking fault-tolerance) Given two colored Kripke structures $M = \langle S, I, R, L, \mathcal{N} \rangle$ and $M' = \langle S', I', R', L', \mathcal{N}' \rangle$, we say that a relationship $\prec_{Mask} \subseteq S \times S'$ is *masking fault-tolerant* for sublabelings $L_0 \subseteq L$ and $L'_0 \subseteq L'$ iff:

(A) $\forall s_1 \in I : (\exists s_2 \in I' : s_1 \prec_{Mask} s_2)$ and $\forall s_2 \in I' : (\exists s_1 \in I : s_1 \prec_{Mask} s_2)$.

(B) for all $s_1 \prec_{Mask} s_2$ the following holds:

(1) $L_0(s_1) = L'_0(s_2)$.

(2) if $s'_1 \in Post_N(s_1)$, then there exists $s'_2 \in Post(s_2)$ with $s'_1 \prec_{Mask} s'_2$.

(3) if $s'_2 \in Post_N(s_2)$, then there exists $s'_1 \in Post_N(s_1)$ with $s'_1 \prec_{Mask} s'_2$.

(4) if $s'_2 \in Post_F(s_2)$, then either there exists $s'_1 \in Post_N(s_1)$ with $s'_1 \prec_{Mask} s'_2$ or $s_1 \prec_{Mask} s'_2$.

We say that state s_2 is masking fault-tolerant for s_1 when $s_1 \prec_{Mask} s_2$. Intuitively, the intention in the definition is that, starting in s_2, faults can be masked in such a way that the behavior exhibited is the same as that observed when starting from s_1 and executing transitions without faults. Let us explain the above definition. Conditions A, $B.1$, $B.2$ and $B.3$ imply that we have a bisimulation between the normative parts

of M and M', thus the non-faulty behavior of both components is basically the same. Condition $B.4$ states that every outgoing faulty transition from s_2 either must be matched to an outgoing normal transition from s_1, or s'_2 is masking fault-tolerant for s_1; this condition expresses that faulty transitions from the second component mimic a normal behavior of the first component. Finally, it is worth remarking that the condition symmetric to $(B.4)$ is not required, since we are only interested in the masking properties of M'.

Notice that, if there exists a self-loop at state s'_2, then we can stay forever satisfying $s_1 \prec_{Mask} s'_2$. In this case, fairness is an important assumption which allows us to ensure system progress. Another important remark is that an execution could be fair but after a while all its transitions become faulty; in this case we say that the execution diverges by faults. To deal with these executions, we will require that from every state it has to be possible to reach another state where non-faulty transitions are enabled, that is, we always have the possibility in the future of executing a correct action.

Definition 3.1.2 (Fault divergence)**.** We say that a model M does not diverge by faults when for every $s \in S$ there exists $s' \in S$ such that $s \rightarrow^* s'$ and $Post_N(s') \neq \emptyset$. In this case we say that M is a NDF (non-divergent by faults) structure.

That is, a model diverges by faults when it can reach a state where all the actions that can be executed in the future are faulty. The assumption that a model does not diverge by faults is natural in fault-tolerance where assumptions about the way that faults occur are needed to prove properties about systems. In the case of masking fault-tolerance, which is one of the most benign forms of fault-tolerance, the hypothesis that normal actions are not always neglected by the model being analyzed is required, in particular, to ensure the preservation of liveness properties. Note that this condition can be checked with a depth-first search over the model. Other authors, for instance [Arora and Gouda, 1993; Arora and Kulkarni, 1998a,b], require that only a finite number of faults should occur in any execution of the system in order to provide masking fault-tolerance; note that this requirement is stronger than the absence of fault divergence.

Roughly speaking, we say that M' masks faults for M iff for every initial state s_0 of M there exists an initial state s'_0 of M' such that $s_0 \prec_{Mask} s'_0$, for some masking fault-tolerant relation $\prec_{Mask}$; we denote this situation by $M \prec_{Mask} M'$. Let us now

present a simple example to illustrate the above definition.

Example 3.1.1. Let us consider a memory cell that stores a bit of information and supports reading and writing operations. A state in this system maintains the current value of the memory cell ($m = i$, for $i = 0, 1$), writing allows one to change this value, and reading returns the stored value.

A potential fault in this scenario occurs when a cell unexpectedly loses its charge, and its stored value turns into another one (e.g., it changes from 1 to 0 due to charge loss). A typical technique to deal with this situation is *redundancy*: use three memory bits instead of one. Writing operations are performed simultaneously on the three bits. Reading, on the other hand, returns the value that is repeated at least twice in the memory bits; this is known as *voting*, and the value read is written back to the three bits.

We take the following approach to model this system: each state is described by variables m and w, which record the value stored in the system (taking *voting* into account) and the last writing operation performed, respectively. The state also maintains the values of the three bits that constitute the system, captured by boolean variables c_0, c_1 and c_2. For instance, in Figure 3.1, state s_0 contains the information 11/111, representing the state: $w = 1$, $m = 1$, $c_0 = 1$, $c_1 = 1$, and $c_2 = 1$.

Consider the colored Kripke structures M (left) and M' (right) depicted in Figure 3.1. M only contains normal transitions describing the expected ideal behavior (without taking into account faults). M' includes a model of a fault: a bit may suffer a discharge and then it changes its value from 1 to 0.

We can show that in this simple case there exists a relation of masking fault-tolerance between M and M' with the sublabelings L_0 and L'_0 obtained by restricting L and L' to propositions m and w, respectively. The relation

$$R_1 = \{\langle s_0, t_0\rangle, \langle s_1, t_1\rangle, \langle s_0, t_2\rangle\}$$

is masking fault-tolerant for $\langle M, M'\rangle$. Notice that each pair of R_1 satisfies each condition of Definition 3.1.1:

- $\langle s_0, t_0\rangle$ satisfies condition $B.1$ because $L_0(s_0) = L'_0(t_0)$. Conditions $B.2$ and $B.3$ are satisfied because the transition $s_0 \to s_0$ is masked by $t_0 \to t_0$ and vice versa with $\langle s_0, t_0\rangle \in R_1$, and transition $t_0 \to t_1$ masks $s_0 \to s_1$ and vice versa with

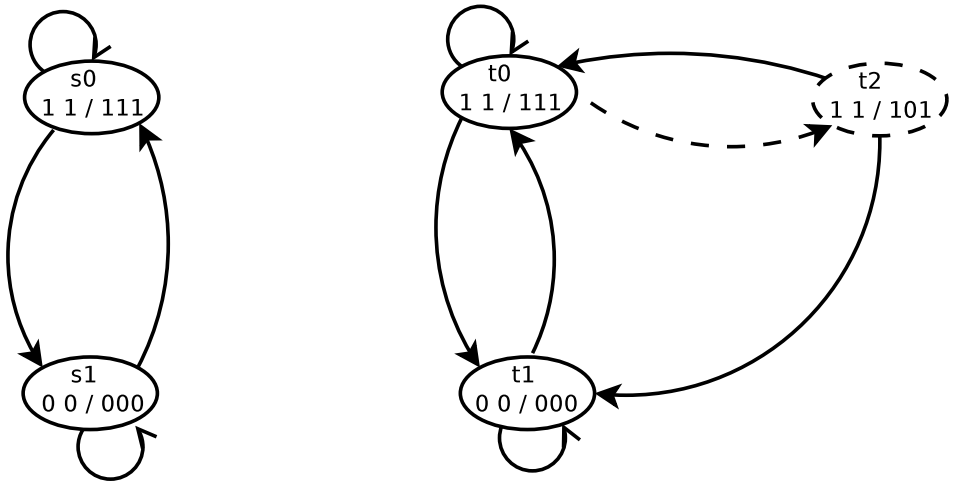

Figure 3.1: Two masking fault-tolerance colored Kripke structures.

$\langle s_1, t_1 \rangle \in R_1$. Finally, condition $B.4$ is satisfied because the faulty transition $t_0 \dashrightarrow t_2$ masks the normal transition $s_0 \rightarrow s_0$ with $\langle s_0, t_2 \rangle \in R_1$.

- $\langle s_1, t_1 \rangle$ satisfies condition $B.1$ because $L_0(s_1) = L'_0(t_1)$. Conditions $B.2$ and $B.3$ are satisfied because the normal transition $t_1 \rightarrow t_0$ is masked by $s_1 \rightarrow s_0$ and vice versa with $(s_0, t_0) \in R_1$, and the normal transition $t_1 \rightarrow t_1$ masks $s_1 \rightarrow s_1$ and vice versa with $\langle s_1, t_1 \rangle \in R_1$.

- $\langle s_0, t_2 \rangle$ satisfies condition $B.1$ since $L_0(s_0) = L'_0(t_2)$ (taking into account that reading operations return the value that is repeated at least twice in the memory bits). Conditions $B.2$ and $B.3$ are satisfied because the normal transition $t_2 \rightarrow t_0$ masks $s_0 \rightarrow s_0$ and vice versa with $\langle s_0, t_0 \rangle \in R_1$, and the normal transition $t_2 \rightarrow t_1$ masks $s_0 \rightarrow s_1$ and vice versa with $\langle s_1, t_1 \rangle \in R_1$.

Now, we provide some results that allows us to apply the definition of masking fault-tolerance to paths.

Lemma 3.1.1. *Let $M = \langle S, I, R, L, \mathcal{N} \rangle$ and $M' = \langle S', I', R', L', \mathcal{N}' \rangle$ be colored Kripke structures, $\prec_{Mask} \subseteq M \times M'$ a masking relation and $s \prec_{Mask} s'$ with $s \in S$ and $s' \in S'$. If M' is a NDF structure, then for any path $\sigma' \in \mathcal{FT}(M')$ such that $\sigma' = s'_0 s'_1 s'_2 \ldots$ with $s'_0 = s'$, there exists a function $f : \mathbb{N} \rightarrow \mathbb{N}$ and a normative path $\sigma \in \mathcal{FNT}(M)$ such that $\sigma = s_{f(0)} s_{f(1)} s_{f(2)} \ldots$ with $s_0 = s$ and $s_{f(i)} \prec_{Mask} s'_i$. Furthermore, f preserves initial segments of $\mathbb{N}$.*

Proof. Note that a path in S is just a function $\sigma : \mathbb{N} \to S$ that respects the transitions in the structure. Given $\sigma' \in \mathcal{FT}(M')$, we define a trace $\sigma^* : \mathbb{N} \to S \cup \{*\}$ where $*$ is a new state, as follows:

$$\sigma^*[i] = \begin{cases} s & \text{if } i = 0, \\ w & \text{if } \exists w \in Post_N(\sigma^*[last(\sigma^*, i)]) \text{ s.t. } w \prec_{Mask} \ s'_i \text{ and }, \\ * & \text{otherwise.} \end{cases}$$

Where $last(\sigma^*, i) = max\{j | 0 \leq j < i : \sigma^*(j) \neq *\}$, that is, this expression denotes the last position in σ^* that is less than i and contains an element different from $*$. Moreover, we choose some $w \prec_{Mask} \ s'_i$; if we have several states that satisfy this condition, we define the function in such a way that it selects the state that has the minimum number of occurrences in $\sigma^*[0..i-1]$. This ensures the fairness of the execution.

Function f is defined by means an auxiliary function $g(i, j)$, which calculates the position of the j-th symbol different from $*$ from position i in σ^*.

$$g(i,j) = \begin{cases} j & \text{if } i = 1 \text{ and } \sigma^*[j] \neq *; \\ g(i-1, j+1) & \text{if } i \neq q \text{ and } \sigma^*[j] \neq *; \\ g(i, j+1) & \text{if } \sigma^*[i] = *. \end{cases}$$

Then, f is defined as follows:

$$f(i) = g(i+1, 0)$$

Now, we f to define a normative trace σ as follows:

$$\sigma[i] = \sigma^*[f(i)]$$

Notice that in the last item in the definition of g the $*$ symbol is skipped; this function is well-defined because we cannot have an infinite sequence of $*'s$ in σ^*. This is straightforward to prove by using the fact that σ' is a fair execution and M' does not diverge by faults; thus, in any infinite execution we have an infinite number of non-faulty actions enabled.

By definition $\sigma \in \mathcal{NT}(M)$, and $s_{f(i)} \prec_{Mask} \ s'_i$. Furthermore, note that σ is a fair execution; if we have that a transition $s_i \to t$ is enabled infinitely often in σ, then we have (by condition $B.2$) an infinite number of positions k in σ' such that t

$\prec_{Mask}\ \sigma'[k]$. Then, by definition of σ^*, at some point we will choose t as successor of some $\sigma[i]$, and this will happen infinitely often, thus $\sigma \in \mathcal{FNT}(M)$. Note that, if $[0 \dots i]$ is an initial segment of $\mathbb{N}$, then, by definition of f, $f([0 \dots i])$ (its image by f) is also an initial segment of $\mathbb{N}$. □

Notice that, if we only consider normal executions starting in s_1 and s_2 with $s_1 \prec_{Mask}\ s_2$, by conditions $(B.3)$ and $(B.4)$ of definition 3.1.1 we have that, for each normative path starting in s_2, there exists a corresponding path from s_1, where its states are similar by masking. This is proven in the following lemma.

Lemma 3.1.2. *Let $M = \langle S, I, R, L, \mathcal{N} \rangle$ and $M' = \langle S', I', R', L', \mathcal{N}' \rangle$ be colored Kripke structures, $\prec_{Mask} \subseteq M \times M'$ a masking relation and $s \prec_{Mask}\ s'$ with $s \in S$ and $s' \in S'$. Then there exist functions $f : \mathcal{FNT}(M)(s) \to \mathcal{FT}(M')(s')$ and $g : \mathcal{FNT}(M')(s') \to \mathcal{FNT}(M)(s)$ such that:*

- $\forall \sigma \in \mathcal{FNT}(M)(s) : \forall i \geq 0 : \sigma[i] \prec_{Mask}\ f(\sigma)[i]$,
- $\forall \sigma' \in \mathcal{FNT}(M')(s') : \forall i \geq 0 : g(\sigma)[i] \prec_{Mask}\ \sigma[i]$

Proof. We prove the first item; the other one is similar. Given $\sigma \in \mathcal{FNT}(M)(s)$ s.t. $\sigma = s s_1 s_2 \dots$, then note that we have $s \prec_{Mask}\ s'$ and by $(B.2)$, if $s_i \prec_{Mask}\ s'_i$ and $s_{i+1} \in Post_N(s_i)$ then there exists $s'_{i+1} \in Post(s'_i)$ such that $s_{i+1} \prec_{Mask}\ s'_{i+1}$. Thus, we can define inductively a sequence $f(\sigma) = s' s'_1 s'_2 \dots$ that satisfies $\sigma[j] \prec_{Mask}\ f(\sigma)[j]$ for every j. To ensure the fairness of such a construction, when choosing the successor of a given $f(\sigma)[i]$ where $\sigma[i] \prec_{Mask}\ f(\sigma)[i]$, we select the successor $f(\sigma)[i] \to t$ such that $\sigma[i+1] \prec_{Mask}\ t$ which appears the minimum number of times in $f(\sigma)[0..i]$, which exists by condition $(B.2)$. The result follows. □

Now, we can prove that, in the case of masking simulation, the liveness and safety properties of the normal behavior of the specification are preserved by the implementation. However, let us note that not all the temporal properties are preserved; formulas with occurrences of the next operator may not be preserved by the implementation. Roughly speaking, this is because condition $(B.4)$ allows implementations to advance in time while staying on the same state in the specification side.

Theorem 3.1.3. *Let $M = \langle S, I, R, L, \mathcal{N} \rangle$ and $M' = \langle S', I', R', L', \mathcal{N}' \rangle$ be colored Kripke structures, $s_1 \in S$ and $s_2 \in S'$. If $s \prec_{Mask}\ s'$ for sublabelings L_0 and L'_0*

obtained by restricting L and L' to AP', respectively, then:

$$M, s \models_{nf} \varphi^N \Leftrightarrow M', s' \models_f \varphi,$$

where φ is a CTL formula with no next operators and all the propositional variables of φ are in AP'.

Proof. We proceed by induction over the structure of the formula φ.

- Base case: $\varphi = p$. From $s \prec_{Mask} s'$ it follows by condition $(B.1)$ for masking fault-tolerance that s and s' have the same valuation. Thus, $M, s \vDash_f p^N \Leftrightarrow M', s' \vDash_f p$, where $(p_i)^N = p_i$ by Definition 2.3.1.

- Inductive case:

Case 1: For $\varphi = \psi \wedge \psi'$ and $\varphi = \neg\psi$ the result follows by applying the inductive hypothesis.

Case 2: $\varphi = \mathsf{A}(\psi_1 \mathcal{U} \psi_2)$. Suppose that $M, s \models_{nf} (\mathsf{A}(\psi_1 \mathcal{U} \psi_2))^N$ and $M', s' \not\models_f \mathsf{A}(\psi_1 \mathcal{U} \psi_2))$. This means that both:

- $\exists \sigma' \in \mathcal{FT}(M') : \exists j \geq 0 : M', \sigma'[j] \not\models \psi_1$ and $\forall 0 \leq i \leq j : M', \sigma'[i] \not\models \psi_2$,
- $\exists \sigma' \in \mathcal{FT}(M') : \forall 0 \leq i : M', \sigma'[i] \not\models \psi_2$.

In the first case, by Lemma 3.1.1 we have a normative path $\sigma \in \mathcal{FNT}(M)$ and function f s.t. $\sigma[f(i)] \prec_{Mask} \sigma[i]$ for any $i \geq 0$, and then by induction we get $M, \sigma[f(i)] \not\models_{nf} (\psi_1)^N$ and also $\forall 0 \leq j \leq i : M, \sigma[f(j)] \not\models_{nf} (\psi_2)^N$, since $f(0), f(1), \ldots, f(i)$ is an initial segment of $\mathbb{N}$ we get that $\forall 0 \leq j \leq f(i) : M, \sigma[j] \not\models \psi_2$, thus $M, \sigma \not\models O((\psi_1)^N \mathcal{U} (\psi_2)^N)$. The second case is similar. The other direction can be proven similarly but using Lemma 3.1.2.

Case 3: $\varphi = \mathsf{E}(\psi_1 \mathcal{U} \psi_2)$. Suppose that $M, s \models_{nf} (\mathsf{E}(\psi_1 \mathcal{U} \psi_2))^N$, then for some path $\sigma \in \mathcal{FNT}(M)$ with $\sigma[0] = s$ we have that:

$$\exists i \geq 0 : M, \sigma[i] \models_{nf} (\psi_2)^N \text{ and } \forall 0 \leq j \leq i : M, \sigma[j] \models_{nf} (\psi_1)^N$$

then by Lemma 3.1.2 we have a path $f(\sigma) \in \mathcal{FT}(M')$ with $f(\sigma)[0] = s'$ s.t. $\forall i \geq 0 : \sigma[i] \prec_{Mask} f(\sigma)[i]$, this implies by induction that:

$$\exists i \geq 0 : M', f(\sigma)[i] \models_f \psi_2 \text{ and } \forall 0 \leq j \leq i : M, f(\sigma)[j] \models_f \psi_1$$

then, $M', s' \models_f \mathsf{E}(\psi_1 \,\mathcal{U}\, \psi_2)$. The other direction is similar.

Case 4: The cases $\varphi = \mathsf{A}(\psi_1 \,\mathcal{W}\, \psi_2)$ and $\varphi = \mathsf{E}(\psi_1 \,\mathcal{W}\, \psi_2)$ are similar to cases 2 and 3, respectively.

□

Summing up, in the case of masking simulation, the basic temporal properties of systems without faults, such as invariants or liveness formulas, are preserved by implementations with faults.

3.2 Nonmasking Fault-tolerance

We now focus on nonmasking fault-tolerance. This kind of tolerance is more permissive than masking tolerance; recall that it allows for the existence of some states that do not mask faults. Intuitively, this type of fault-tolerance allows the system to violate its specification while it is recovering from a fault and thus returning to a normal behavior. More technically, the liveness properties of the nonfaulty part of the system are preserved, whereas the safety properties observed in the correct behavior of the system may not be fully preserved, but should be *eventually* reinstated. The characterization of this kind of fault-tolerance is as follows.

Definition 3.2.1. (Nonmasking fault-tolerance) Given two colored Kripke structures $M = \langle S, I, R, L, \mathcal{N} \rangle$ and $M' = \langle S', I', R', L', \mathcal{N}' \rangle$, we say that a relation $\prec_{Nonmask} \subseteq S \times S'$ is nonmasking for sublabelings $L_0 \subseteq L$ and $L'_0 \subseteq L'$, iff:

(A) $\forall s_1 \in I : (\exists s_2 \in I' : s_1 \prec_{Nonmask} s_2)$ and $\forall s_2 \in I' : (\exists s_1 \in I : s_1 \prec_{Nonmask} s_2)$.

(B) for all $s_1 \prec_{Nonmask} s_2$ the following holds:

(1) $L_0(s_1) = L'_0(s_2)$.

(2) if $s'_1 \in Post_N(s_1)$, then there exists $s'_2 \in Post(s_2)$ with $s'_1 \prec_{Nonmask} s'_2$.

(3) if $s'_2 \in Post_N(s_2)$, then there exists $s'_1 \in Post_N(s_1)$ with $s'_1 \prec_{Nonmask} s'_2$.

(4) if $s'_2 \in Post_F(s_2)$, then there exists $s'_1 \in Post_N(s_1)$ with $s'_1 \prec_{Nonmask} s'_2$, or

(5) if $s'_2 \in Post_F(s_2)$ with $s'_1 \not\prec_{Nonmask} s'_2$ for all $s'_1 \in Post_N(s_1)$, then for any finite fragment $s_2 s'_2 w_0 \dots w_k$ such that $s'_1 \not\prec_{Nonmask} w_i$ for all $s'_1 \in Post_N(s_1)$ and w_i, there exists a path $w_k \Rightarrow^* s''_2$ such that $s'_1 \prec_{Nonmask} s''_2$ for some $s'_1 \in Post_N(s_1)$.

Let us briefly explain this definition. Conditions A, $B.1$, $B.2$, $B.3$, $B.4$ are similar to the conditions of Definition 3.1.1. Condition $B.5$ asserts that, if $s_1 \prec_{Nonmask} s_2$ and every "faulty" successor state (say s'_2) of s_2 is not in a nonmasking relation with any normal successor of s_1, then any faulty path fragment starting at s'_2 can be extended to reach a s''_2 such that $s'_1 \prec_{Nonmask} s''_2$ for some normal successor s'_1 of s_1; that is, the system can recover from faults.

We say that M' is nonmasking fault-tolerant with respect to M iff for every initial state s_0 of M there exists an initial state s'_0 of M' such that $s_0 \prec_{Nonmask} s'_0$, for some nonmasking fault-tolerant $\prec_{Nonmask}$ (indicated by $M \prec_{Nonmask} M'$).

At first sight, nonmasking fault-tolerance seems similar to the notion of weak bisimulation used in process algebra [Milner, 1980], where silent steps are taken into account. Notice, however, that, as opposed to weak bisimulation where silent steps produce only non-observable (i.e., internal) changes, faults may produce observable changes in a nonmasking fault-tolerance relation. Let us present an example of nonmasking fault-tolerance.

Example 3.2.1. For the memory cell introduced in Example 3.1.1, consider now the colored Kripke structures M (left) and M' (right) depicted in Figure 3.2. Now we consider that two faults may occur: up to two bits may lose their charge before any normal transition is taken. The relation $R_2 = \{\langle s_0, t_0\rangle, \langle s_1, t_1\rangle, \langle s_0, t_2\rangle\}$ is nonmasking tolerant for $\langle M, M'\rangle$ and the sublabelings L_0 and L'_0, obtained by restricting L and L' to propositions m and w, respectively.

In nonmasking fault-tolerance, one is interested in preserving the liveness properties of the non-faulty part of the system, that is, the requests need to be granted, or rather the program should exhibit some advance towards some goal, even during a faulty scenario. Note that in this case safety conditions do not need to be preserved. As acknowledged in [Gärtner, 1999a], this kind of fault-tolerance is not the usual one

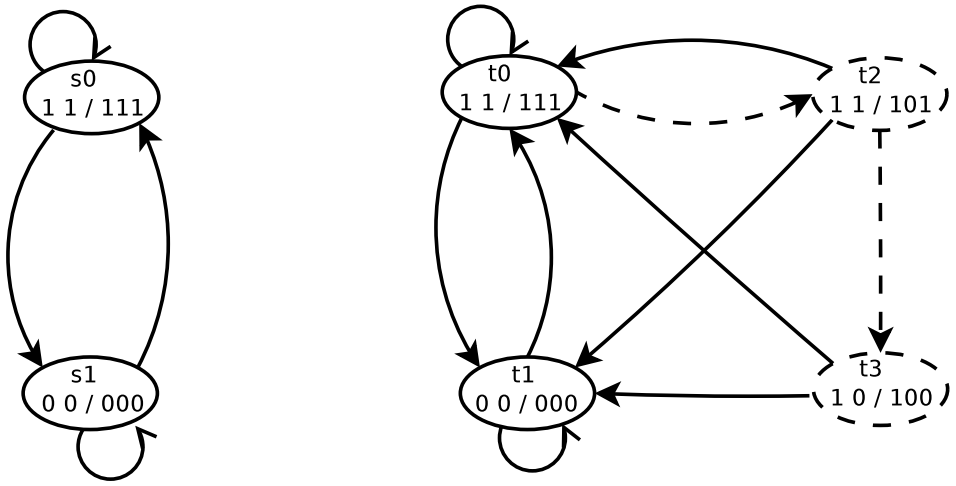

Figure 3.2: Two nonmasking fault-tolerance colored Kripke structures.

in practice since one usually intends to keep the system in safe states. First, we prove that liveness properties are preserved by nonmasking simulation. To do so, we need a couple of lemmas; the following two lemmas relate normative and faulty executions of two systems related by a nonmasking relation.

Lemma 3.2.1. *Let $M = \langle S, I, R, L, \mathcal{N} \rangle$ and $M' = \langle S', I', R', L', \mathcal{N}' \rangle$ be colored Kripke structures, $s \in S$, $s' \in S'$ with $s \prec_{Nonmask} s'$ for some sublabelings L_0 and L'_0 obtained by restricting L and L' to AP'. If there is a $\sigma' \in \mathcal{FT}(M')(s')$ with $M', \sigma' \vDash \mathsf{G}p$ for some $p \in AP'$, then there exists a $\sigma \in \mathcal{FNT}(M)(s)$ such that $M, \sigma \vDash \mathsf{G}p$.*

Proof. Suppose that we have a $\sigma' \in \mathcal{FT}(M')(s')$ such that $M', \sigma' \vDash \mathsf{G}p$. We know that $s \prec_{Nonmask} s'$, thus we can define an execution σ of M and a function $f : \mathbb{N} \to \mathbb{N}$ as follows:

- $\sigma[0] = s$ and $f(0) = 0$,
- Let $\sigma[i]$ be a defined position. Note that we have $\sigma[i] \prec_{Nonmask} \sigma'[f(i)]$, $\sigma[i+1]$ is a $w \in S$ such that $w \in Post_N(\sigma[i])$ and there is a $\sigma'[f(i)+k]$ such that $w \prec_{Nonmask} \sigma'[f(i)+k]$ (for some k). If there are many such w's, we select the one that appears less often in $\sigma[0..i]$. Note that, because of Definition 3.2.1 we have a least one. Furthermore, we define $f(i+1) = f(i) + k$.

First, let us note that since every position of σ is nonmasking similar to a position of σ', we have $M', \sigma' \vDash \mathsf{G}p$. We need to prove that $\sigma \in \mathcal{FNT}(M)(s)$. Suppose

that $\sigma \notin \mathcal{FNT}(M)(s)$, that is we have $w \in Post_N(\sigma[i])$ infinitely often in σ for some $t = \sigma[i]$, and the fragment $\ldots tw \ldots$ does not occur infinitely often in σ. But since σ' is fair, and by definition of σ, we have an unbounded number of positions i where $\sigma'[i] \to t_i$ is a transition in M' and $w \prec_{Nonmask} t_i$. Since we have a finite number of transitions, we have states $t_k, t'_{k'}$ such that $t_k \to t'_{k'}$ is enabled infinitely often in σ', and so (σ' is fair) we have that the fragment $\ldots t_k t'_{k'} \ldots$ occurs infinitely often in σ'. Now, note that fragment $\ldots tw \ldots$ also should occur infinitely often in σ by construction, since at some point we select w as successor since it is the state that occurs a minimum number of time in the sequence σ. That is, we obtain a contradiction and so $\sigma \in \mathcal{FNT}(M)(s)$. □

Similarly we can relate traces of the faulty model with those of the specification.

Lemma 3.2.2. *Given $M = \langle S, I, R, L, \mathcal{N} \rangle$ and $M' = \langle S', I', R', L', \mathcal{N}' \rangle$, $s \in S$, $s' \in S'$ with $s \prec_{Nonmask} s'$ for some for sublabelings L_0 and L'_0 obtained by restricting L and L' to AP'. If there is a $\sigma \in \mathcal{FNT}(M)(s)$, such that $M, \sigma \vDash \mathsf{F}p$, for some p in L_0, then there is a $\sigma' \in \mathcal{FT}(M')(s')$ such that $M', \sigma' \vDash \mathsf{F}p$.*

Proof. Given $\sigma \in \mathcal{FNT}(M)(s)$, we define an execution $\sigma' \in \mathcal{TR}(M')$ as follows:

- $\sigma'[0] = s'$,
- suppose that $\sigma'[i]$ has been already defined, $\sigma'[i+1]$ is the state w such that $\sigma[i] \Rightarrow^* w$ and $\sigma[i] \prec_{Nonmask} w$, that appears the minimum number of times in $\sigma'[0..i]$.

As in Lemma 3.2.1, the definition of σ' guarantees that $\sigma' \in \mathcal{FT}(M')s'$. Now, if we have some k such that $M, \sigma[k] \vDash p$ then, we have that $\sigma[k] \prec_{Nonmask} \sigma'[k]$, and then $M', \sigma' \vDash \mathsf{F}p$, the result follows. □

Now, we can prove that, in the presence of fairness, liveness properties are preserved by nonmasking implementations:

Theorem 3.2.3. *Let $M = \langle S, I, R, L, \mathcal{N} \rangle$ and $M' = \langle S', I', R', L', \mathcal{N}' \rangle$ be colored Kripke structures $s \in S$ and $s' \in S'$. If $s \prec_{Nonmask} s'$ for sublabelings L_0 and L'_0 obtained by restricting L and L' to AP', respectively, then*

$$M, s \vDash_{nf} \varphi^N \Rightarrow M', s' \vDash_f \varphi,$$

where all the propositional variables of φ are in AP', and φ is a liveness CTL property.

Proof. The proof is by induction on φ. For the base case, we have two possibilities:

- If $\varphi = \mathsf{EF}p$, suppose $M, s \vDash_{nf} \varphi^N$, that is, there is an execution $\sigma \in \mathcal{FNT}(M)$ such that $M, \sigma[i] \vDash p$, for some i, thus by Lemma 3.2.1, we have a $\sigma' \in \mathcal{FT}(M')$ such that $M', \sigma'[k] \vDash p$ for some k.

- IF $\varphi = \mathsf{AF}p$, we reason similarly to the case above, but using Lemma 3.2.2.

The inductive cases (that is, $\psi_1 \vee \psi_2$ and $\psi_1 \wedge \psi_2$) are direct using the inductive hypothesis. □

Note that this theorem guarantees that implementations preserve the liveness properties of specifications. Furthermore, notice that the other direction of this property is not necessarily true. This is mainly because nonmasking implementations may eventually make true some properties, during its faulty behavior, which do not hold in the non-faulty program.

As argued in [Arora and Kulkarni, 1998b; Gärtner, 1999a], in practice we are interested in those nonmasking programs that eventually reestablish the safety properties of their specifications, that is, faulty programs may exhibit an incorrect behavior, but at some point they start behaving as expected. Obviously, to guarantee such a property in a nonmasking simulation, we need to avoid such scenarios where faults occur in such a way that the system cannot reach a normal execution. When all the executions of the system only exhibit a finite number of faults, then we can ensure that the normal behavior of the system will be reestablished; this is proven in the following theorem.

Theorem 3.2.4. *Let $M = \langle S, I, R, L, \mathcal{N} \rangle$ and $M' = \langle S', I', R', L', \mathcal{N}' \rangle$ be colored Kripke structures such that $s \in S$ and $s' \in S'$ and for any $\sigma' \in \mathcal{FT}(M')(s')$ the number of i's such that $\sigma'[i+1] \in Post_F(\sigma'[i])$ is finite. Then, if $s \prec_{Nonmask} s'$ for sublabelings L_0 and L'_0 obtained by restricting L and L' to AP', respectively, then*

$$M, s \vDash_{nf} (\mathsf{AG}\varphi)^N \Rightarrow M', s' \vDash_f \mathsf{AFAG}\varphi,$$

where all the propositional variables of φ are in AP'.

Proof. To prove this property, we need to take note of some observations. First, note that, since we have finite structures, requiring that the number of faults of any execution is bounded, is the same as requiring that no faults occur in any cycle, otherwise we can find a trace (the cycle) where we have an unbounded number of faults. That is, in this case, for any execution $\sigma' \in \mathcal{TR}(M')$, we have an instant i such that $\mathcal{TR}(M')(\sigma'[i]) \subseteq \mathcal{NT}(M')(\sigma'[i])$, that is, we have a point from which all the transitions are non-faulty. Note also that, if $s \prec_{Nonmask} s'$ with $\mathcal{TR}(M)(s) \subseteq \mathcal{NT}(M)(s)$ and $\mathcal{TR}(M')(s') \subseteq \mathcal{NT}(M')s'$, then we have that s and s' are bisimilar (there are no faulty actions, and $B.2$ and $B.3$ for Definition 3.2.1 guarantee a bisimulation) and thus $M, s \vDash \varphi$ iff $M', s' \vDash \varphi$ for any CTL formula φ. Now, suppose that $M, s \vDash_{nf} (\mathsf{AG}\varphi)^N$. Take any path $\sigma' \in \mathcal{TR}(M')s'$, at some point i; we have that $\mathcal{TR}(M')\sigma'[i] \subseteq \mathcal{NT}(M')\sigma'[i]$. Also by Definition 3.2.1 we have instants k and $k' > i$ such that $\sigma[k] \prec_{Nonmask} \sigma[k']$, and also note that, by hypothesis, we have $M, \sigma[k] \vDash_{nf} (\mathsf{AG}\varphi)^N$. By the remark above, we have $M', \sigma'[k'] \vDash_f \mathsf{AG}\varphi$ and thus $M', s \vDash \mathsf{AFAG}\varphi$. □

Summarizing, this theorem says that, if we only consider a finite number of faults occurring in executions, then safety properties are eventually reestablished.

3.3 Failsafe Fault-tolerance

We now present a characterization of failsafe fault-tolerance. Essentially, failsafe fault-tolerance must ensure that the system will stay in a safe state, although it may be limited in its capacity. More technically, this means that the normative safety properties must be preserved, while normative liveness properties may not be respected.

Definition 3.3.1. (Failsafe fault-tolerance) Given two colored Kripke structures $M = \langle S, I, R, L, \mathcal{N}\rangle$ and $M' = \langle S', I', R', L', \mathcal{N}'\rangle$, we say that a relation $\prec_{Failsafe} \subseteq S \times S'$ is failsafe for sublabelings $L_0 \subseteq L$ and $L'_0 \subseteq L'$ iff:

(A) $\forall s_1 \in I : (\exists s_2 \in I' : s_1 \prec_{Failsafe} s_2)$ and $\forall s_2 \in I' : (\exists s_1 \in I : s_1 \prec_{Failsafe} s_2)$.

(B) for all $s_1 \prec_{Failsafe} s_2$ the following holds:

(1) $L_0(s_1) = L'_0(s_2)$.

(2) if $s'_1 \in Post_N(s_1)$, then there exists $s'_2 \in Post(s_2)$ with $s'_1 \prec_{Failsafe} s'_2$.

(3) if $s'_2 \in Post_N(s_2)$, then there exists $s'_1 \in Post_N(s_1)$ with $s'_1 \prec_{Failsafe} s'_2$.

(4) if $s'_2 \in Post_F(s_2)$, then either:

i. there exists $s'_1 \in Post_N(s_1)$ with $s'_1 \prec_{Failsafe} s'_2$ or $s_1 \prec_{Failsafe} s'_2$, or

ii. $\forall s : (s'_2 \Rightarrow^* s) \Rightarrow L_0(s_2) = L_0(s)$

Whenever two states s_1 and s_2 are related by a failsafe fault-tolerant relation $\prec_{Failsafe}$, i.e., $s_1 \prec_{Failsafe} s_2$, we say that s_2 is failsafe fault-tolerant for s_1. We say that M' is failsafe fault-tolerant for M if we have some relation $\prec_{Failsafe} \subseteq S \times S'$; we denote this situation by $M \prec_{Failsafe} M'$.

Let us briefly explain this definition. Conditions A, $B.1$, $B.2$, $B.3$, $B.4.i$ are similar to those conditions of Definition 3.1.1 regarding masking fault-tolerance. Condition $B.4.ii$ intuitively asserts that, from s'_2 the system in M' remains in a state (or several equivalent states) which is safe. We now present a simple example to illustrate this notion.

Example 3.3.1. Consider the colored Kripke structures M (left) and M' (right) depicted in Figure 3.3. M is the specification of the expected ideal, fault-free, behavior. M', on the other hand, involves the occurrence of one fault. The relation $R_3 = \{(s_0, t_0), (s_1, t_1)\}$ is a failsafe fault-tolerance relation for (M, M') and the sublabelings that are obtained by restricting L and L' to propositions m and w.

In the following we prove that our definition of failsafe fault-tolerance preserves safety properties. First, note that a failsafe relation imposes a relationship between the traces of the two models involved in the relations; this is proven in the following lemmas.

Lemma 3.3.1. *Let $M = \langle S, I, R, L, \mathcal{N}\rangle$ and $M' = \langle S', I', R', L', \mathcal{N}'\rangle$ be colored Kripke structures, $s \in S$, $s' \in S'$ with $s \prec_{Failsafe} s'$ for some sublabelings L_0 and L'_0 obtained by restricting L and L' to AP'. If there is a $\sigma' \in \mathcal{FT}(M')(s')$ with $M', \sigma' \vDash \mathsf{F}p$ for some $p \in AP'$, then there exists a $\sigma \in \mathcal{FNT}(M)(s)$ such that $M, \sigma \vDash \mathsf{F}p$.*

Proof. Given $\sigma' \in \mathcal{FT}(M')(s')$ where we have $s \prec_{Failsafe} \sigma'[0]$ and we have $p \in L'_0(\sigma'[k])$ for some k, now we have some fragment $ss_1s_2s_3 \ldots s_{k'}$ in M (by Definition 3.3.1) such that $k \leq k'$ and $\sigma'[i] \prec_{Failsafe} s_j$ for $0 \leq i \leq k$ and $0 \leq j \leq k'$; obviously,

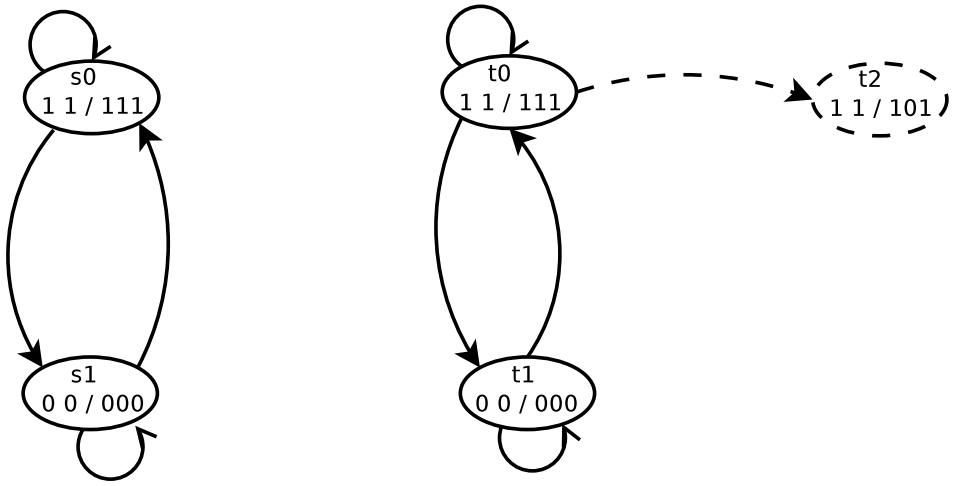

Figure 3.3: Two failsafe fault-tolerant colored Kripke structures.

this segment has some extension σ in M that is a fair execution, and also by Definition 3.3.1 we have that for some i $M, s_i \vDash p$, thus $M, \sigma \vDash \mathsf{F}p$ □

Lemma 3.3.2. *Let $M = \langle S, I, R, L, \mathcal{N} \rangle$ and $M' = \langle S', I', R', L', \mathcal{N}' \rangle$ be colored Kripke structures, $s \in S$, $s' \in S'$ with $s \prec_{Failsafe} s'$ for some for sublabelings L_0 and L'_0 obtained by restricting L and L' to AP'. If there is a $\sigma \in \mathcal{FNT}(M)(s)$ with $M, \sigma \vDash \mathsf{G}p$ for some $p \in AP'$, then there exists a $\sigma' \in \mathcal{FT}(M')(s')$ such that $M', \sigma' \vDash \mathsf{G}p$.*

Proof. Given $\sigma \in \mathcal{FNT}(M')(s')$ such that $M, s \vDash \mathsf{G}p$, then we can define an execution σ' in $\mathcal{FT}(M')(s')$ similarly to the case in Lemma 3.2.2, thus by Definition 3.3.1 we have $M', \sigma'[i] \vDash p$ for all i, then $M', \sigma' \vDash \mathsf{G}p$. □

Let us now prove that failsafe implementations preserve safety properties.

Theorem 3.3.3. *Let $M = \langle S, I, R, L, \mathcal{N} \rangle$ and $M' = \langle S', I', R', L', \mathcal{N}' \rangle$ be colored Kripke structures, $s_1 \in S$ and $s_2 \in S'$. If $s_1 \prec_{Failsafe} s_2$ for sublabelings L_0 and L'_0 obtained by restricting L and L' to AP', respectively, and φ is a CTL safety property. Then $M, s_1 \vDash_{nf} \varphi^N \Rightarrow M', s_2 \vDash_f \varphi$, where all the propositional variables of φ are in AP'.*

Proof. We proceed by induction on φ. The base case is as follows:

- If $\varphi = \mathsf{AG}p$, then suppose $M, s \vDash_{nf} (\mathsf{AG}p)^N$ and not $M', s' \vDash_f \mathsf{AG}p$. Thus we have a $\sigma' \in \mathcal{FT}(M')(s')$ such that $M', \sigma' \vDash \mathsf{F}p$. So by Lemma 3.3.1 we have $M, \sigma \vDash \mathsf{F}p$ for $\sigma \in \mathcal{FNT}(M)(s)$; this is a contradiction and then $M', s' \vDash_f \mathsf{AG}p$.

- If $\varphi = \mathsf{EG}p$, we proceed as before but using Lemma 3.3.2.

The inductive cases are direct by using the inductive hypothesis. □

That is, if we have a failsafe relation between s_1 and s_2 and for every state in normal paths starting in s_1, φ holds in the absence of faults, then φ is always true even in the presence of faults in paths starting in s_2.

3.4 Some Properties

The following lemma presents some properties of all the fault-tolerance relations defined above.

Lemma 3.4.1. *Given relations $\prec_{Mask}$, $\prec_{Nonmask}$ and $\prec_{Failsafe}$, we have the following properties:*

- $\prec_{Mask}$, $\prec_{Nonmask}$, $\prec_{Failsafe}$ *are transitive,*
- *If M does not have faults, then: $M \sqsubset M' \Rightarrow M' \sqsubset M$, where $\sqsubset \in \{\prec_{Mask}, \prec_{Nonmask}, \prec_{Failsafe}\}$,*
- $\prec_{Mask}$, $\prec_{Nonmask}$ *and* $\prec_{Failsafe}$ *are not necessarily reflexive.*

Proof. We prove the properties for $\prec_{Mask}$. The proofs for the other relations are similar.

- **Nonreflexivity:** We show that $\prec_{Mask}$ is not reflexive via the following counterexample: given the colored Kripke structure M with state space $S = \{s_0, s_1\}$ depicted in Figure 3.4. Its identity relation $R = \{\langle s_0, s_0\rangle, \langle s_1, s_1\rangle\}$ is not masking fault-tolerant because the pair $\langle s_0, s_0\rangle$ does not satisfy condition $(B.4)$ of Definition 3.1.1: for $s_1 \in Post_F(s_0)$, there does not exist a normal successor from s_0 with $s_1 \prec_{Mask} s_1$.

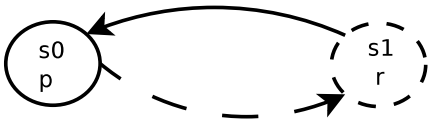

Figure 3.4: Counterexample for reflexivity.

- **Symmetry**: Assume R is masking fault-tolerant for two colored Kripke structures M and M'. Clearly, relation R^{-1} satisfies conditions (A) and $(B.1)$. Conditions $(B.2)$ and $(B.3)$ also hold for R^{-1} by symmetry of $(B.2)$ and $(B.3)$. Finally, condition $(B.4)$ is satisfied due to the fact that we do not take in account the faulty transitions in the left colored Kripke structures of the pair $\langle M, M' \rangle$. Hence, R^{-1} is masking fault-tolerant for $\langle M', M \rangle$.

- **Transitivity** Let $R_{1,2}$ and $R_{2,3}$ be masking fault-tolerant for $\langle M_1, M_2 \rangle$ and $\langle M_2, M_3 \rangle$, respectively. The relation $R_{1,3} = R_{1,2} \circ R_{2,3}$, defined as usual, is masking fault-tolerant for $\langle M_1, M_3 \rangle$, where S_2 denotes the set of states in M_2. This can be proven by checking the conditions of Definition 3.1.1:

  (A) Consider the initial state s_1 of M_1. Since $R_{1,2}$ is masking fault-tolerant, there is an initial state s_2 of M_2 with $\langle s_1, s_2 \rangle \in R_{1,2}$. As $R_{2,3}$ is masking fault-tolerant, there is an initial state s_3 of M_3 with $\langle s_2, s_3 \rangle \in R_{2,3}$. Thus, $\langle s_1, s_3 \rangle \in R_{1,3}$. In the same way, we can check that for any initial state s_3 of M_3, there is an initial state s_1 of M_1 with $\langle s_1, s_3 \rangle \in R_{1,3}$.

  (B.1) By definition of $R_{1,3}$, there is a state s_2 in M_2 with $\langle s_1, s_2 \rangle \in R_{1,2}$ and $\langle s_2, s_3 \rangle \in R_{2,3}$. Then, $L_1(s_1) = L_2(s_2) = L_3(s_3)$.

  (B.2) Assume $\langle s_1, s_3 \rangle \in R_{1,3}$. As $\langle s_1, s_2 \rangle \in R_{1,2}$, it follows that if $s_1' \in Post_N(s_1)$, then $\langle s_1', s_2' \rangle \in R_{1,2}$ for some $s_2' \in Post_N(s_2)$. Since $\langle s_2, s_3 \rangle \in R_{2,3}$, we have $\langle s_2', s_3' \rangle \in R_{2,3}$ for some $s_3' \in Post_N(s_3)$. Hence, $\langle s_1, s_3 \rangle \in R_{1,3}$

  (B.3) Similar to the proof for *(B.2)*.

  (B.4) Assume $\langle s_1, s_3 \rangle \in R_{1,3}$. As $\langle s_1, s_2 \rangle \in R_{1,2}$, it follows that if $s_2' \in Post_F(s_2)$, then $\langle s_1', s_2' \rangle \in R_{1,2}$ for some $s_1' \in Post_N(s_1)$. Since $\langle s_2, s_3 \rangle \in R_{2,3}$, we have $\langle s_2'', s_3' \rangle \in R_{2,3}$ for some $s_3' \in Post_F(s_3)$ and $s_2'' \in Post_N(s_2)$. Moreover, for $s_2 \rightarrow s_2''$, by condition $(B.3)$ there exists $s_1'' \in Post_N(s_1)$ with $\langle s_1'', s_2'' \rangle \in R_{1,2}$. Thus, we have that there exists s_2'' such that $\langle s_1'', s_2'' \rangle \in R_{1,2}$ and $\langle s_2'', s_3' \rangle \in R_{2,3}$. Then, for $s_3' \in Post_F(s_3)$ there exists $s_1'' \in Post_N(s_1)$ with $\langle s_1'', s_3' \rangle \in R_{1,3}$.

□

We also have properties of these relations corresponding to *inclusions*:

Theorem 3.4.2. *Let Mask, NMask and FSafe be the sets of masking, nonmasking and failsafe relations between two colored Kripke structures M and M', and M' a NDF structure, then we have:*

$$\mathsf{Mask} \subseteq \mathsf{FSafe} \textit{ and } \mathsf{Mask} \subseteq \mathsf{NMask}$$

Proof. The inclusion $\mathsf{Mask} \subseteq \mathsf{FSafe}$ is straightforward by definition of masking and nonmasking. To prove the other inclusion, first note that conditions A, $B.1$, $B.2$, and $B.3$ are the same in each one of the definitions. Suppose that $s \prec_{Mask} s'$, and $t' \in Post_F(s')$, if there is a $t \in Post_N(s)$ such that $t \prec_{Mask} t'$, then we also have that condition $B.4$ of Definition 3.2.1 holds, and so the relation is also a nonmasking relationship. Now, in the other case we have $s \prec_{Mask} t'$, but since M' is NDF, we can reach a non-faulty action at some point, and thus we have a fragment $t' \Rightarrow^* w$ such that $s' \prec_{Mask} w$ by condition $B.3$. Thus the relation also satisfies condition $B.5$ of nonmasking. □

3.5 Checking Fault-Tolerance Properties

Simulation and bisimulation relations are amenable to efficient computational treatment. In [Baier and Katoen, 2008; Henzinger et al., 1995], algorithms for calculating several simulation and bisimulation relations are described and proved to be of polynomial complexity with respect to the number of states and transitions of the corresponding models. We have adapted these algorithms to our setting, thus obtaining efficient procedures to prove masking, nonmasking and failsafe fault-tolerance. Such algorithms can be used to verify whether $M \sqsubset M'$, with $\sqsubset \in \{\prec_{Mask}, \prec_{Nonmask}, \prec_{Failsafe}\}$. In this section, we present the algorithms for computing masking, nonmasking, and failsafe relations. In general, these algorithms take as input a colored Kripke structure $M = \langle S, I, R, L, \mathcal{N} \rangle$ and a sublabeling $L_0 \subseteq L$, and produce a fault-tolerance relation $\prec_{Mask}$, $\prec_{Nonmask}$, or $\prec_{Failsafe}$, when they exist. In order to check fault-tolerance properties, we take two colored Kripke structures $M = \langle S, I, R, L, \mathcal{N} \rangle$ and $M' = \langle S', I', R', L', \mathcal{N}' \rangle$ over AP (the system specification and the fault-tolerant implementation), and combine them in a single structure $M \oplus M'$ via disjoint union, to feed as input to the corresponding algorithm.

3.5.1 Computing Masking Fault-Tolerance

The basic scheme for checking masking fault-tolerance is sketched in Algorithms 1 and 2. The first algorithm checks condition $B.2$ of Definition 3.1.1 using a standard procedure for computing simulations [Henzinger et al., 1995], whereas the latter takes care of conditions $B.3$ and $B.4$. In this later case some additional steps (as explained below) are needed to deal with faulty transitions.

Algorithm 1 Computation of condition $B.2$ of def. 3.1.1

Require: Colored Kripke structure M with set of states S
Ensure: Relation $\prec_{SimB2}$ where the states in $SimB2$ hold condition $B.2$

1: **for all** $s_1 \in \mathcal{N}$ **do**
2: $SimB2(s_1) := \{s_2 \mid L_0(s_1) = L_0(s_2)\}$
3: $RemoveR(s_1) := S \backslash Pre(SimB2(s_1))$
4: **end for**
5: **while** $\exists\ s'_1 \in \mathcal{N}$ with $RemoveR(s'_1) \neq \emptyset$ **do**
6: select s'_1 such that $RemoveR(s'_1) \neq \emptyset$
7: **for all** $s_2 \in RemoveR(s'_1)$ **do**
8: **for all** $s_1 \in Pre_N(s'_1)$ **do**
9: **if** $s_2 \in SimB2(s_1)$ **then**
10: $SimB2(s_2) := SimB2(s_2) \backslash \{s_1\}$
11: **for all** $s \in Pre(s_2)$ with $Post(s) \cap SimB2(s_1) = \emptyset$ **do**
12: $RemoveR(s_1) := RemoveR(s_1) \cup \{s\}$
13: **end for**
14: **end if**
15: **end for**
16: **end for**
17: $RemoveR(s'_1) := \emptyset$
18: **end while**
19: **return** $\{\langle s_1, s_2 \rangle \mid s_2 \in SimB2(s_1)\}$

First, let us note that Algorithm 1 is also used in the algorithms for computing masking and failsafe relations, since the definitions of these relations also require condition $B.2$. We briefly explain Algorithm 2. For each $s_2 \in S$, the set $Mask(s_2)$ contains the normal states that are candidates for masking s_2.

Initially, $Mask(s_2)$ consists of all normal states with the same labels as s_2 and $Remove(s_2)$ contains all the normal states which do not have a (normal) successor state masking s_2. Moreover, these states cannot mask any of the predecessors of s_2.

Algorithm 2 Computation of masking fault-tolerant

Require: colored Kripke structure M with set of states S

Ensure: Masking fault-tolerance relation $\prec_{Mask}$ for M, where conditions $B.3$ and $B.4$ of definition 3.1.1 hold

1: **for all** $s_2 \in S$ **do**
2: $Mask(s_2) := \{s_1 \in \mathcal{N} \mid L_0(s_1) = L_0(s_2)\}$
3: $Remove(s_2) := \mathcal{N} \backslash Pre_N(Mask(s_2))$
4: **end for**
5: **while** $\exists s_2' \in S$ with $Remove(s_2') \neq \emptyset$ **do**
6: select s_2' such that $Remove(s_2') \neq \emptyset$
7: **for all** $s_1 \in Remove(s_2')$ **do**
8: **for all** $s_2 \in Pre_N(s_2')$ **do**
9: **if** $s_1 \in Mask(s_2)$ **then**
10: $Mask(s_2) := Mask(s_2) \backslash \{s_1\}$
11: **for all** $s \in Pre_N(s_1)$ with $Post_N(s) \cap Mask(s_2) = \emptyset \wedge (s \notin Mask(s_2) \vee s \notin Mask(Pre_F(s_2)))$ **do**
12: $Remove(s_2) := Remove(s_2) \cup \{s\}$
13: **end for**
14: **end if**
15: **end for**(*This takes care of faulty transitions*)
16: **for all** $s_2 \in Pre_F(s_2')$ **do**
17: **if** $s_1 \in Mask(s_2) \wedge s_1 \notin Mask(s_2')$ **then**
18: $Mask(s_2) := Mask(s_2) \setminus \{s_1\}$
19: **for all** $s \in Pre_N(s_1)$ with $Post_N(s) \cap Mask(s_2) = \emptyset$ **do**
20: $Remove(s_2) := Remove(s_2) \cup \{s\}$
21: **end for**
22: **end if**
23: **end for**
24: **end for**
25: $Remove(s_2') := \emptyset$
26: **end while**
27: **return** $\{\langle s_1, s_2 \rangle \mid s_1 \in Mask(s_2) \wedge s_2 \in SimB2(s_1)\}$

The termination condition of the loop of lines 5-26 is $Remove(s'_2) = \emptyset$ for all $s'_2 \in S$, in which case there are no normal states that need to be removed from the sets of simulators $Mask(s_2)$ for $s_2 \in Pre_N(s'_2)$ or $s_2 \in Pre_F(s'_2)$. Within the while-loop body, the main idea is to pick up one pair $\langle s_1, s'_2 \rangle$ with $s_1 \in Remove(s'_2)$ per iteration; for each one we scan through the predecessor list of s'_2 and test for each normal state $s_2 \in Pre_N(s'_2)$ (or $s_2 \in Pre_F(s'_2)$) to see whether $s_1 \in Mask(s_2)$. In the positive case, s_1 is removed from $Mask(s_2)$. Subsequently, we add to the set $Remove(s_2)$ all normal predecessors s of s_1 such that $Post_N(s) \cap Mask(s_2) = \emptyset$, and in the case of faulty transitions, we also check that $s \notin Mask(s_2) \vee s \notin Mask(Pre_F(s_2))$, that is, the last part of condition $B.4$. Finally, the masking fault-tolerance $\prec_{Mask}$ is obtained from a colored Kripke structure M by using the sets obtained from Algorithm 1 and Algorithm 2. The proofs of correctness and termination are obtained by adapting the corresponding proofs given in [Baier and Katoen, 2008].

Theorem 3.5.1 (Partial Correctness of Masking)**.** *On termination, Algorithms 1 and 2 return a relation* $\prec_{Mask} \subset S \times S$.

Proof. We have to prove that Algorithm 2 ensures conditions $B.3$ and $B.4$ of Definition 3.1.1. First, let us note that the loop of line 5 has the following loop invariant. For all state $s_2 \in S$:

(a) $Remove(s_2) \subseteq \mathcal{N} \backslash Pre_N(Mask(s_2))$

(b) for any relation $\prec_{Mask}$: $\{s_1 \in \mathcal{N} \mid s_1 \prec_{Mask} s_2\} \subseteq Mask(s_2) \subseteq \{s_1 \in \mathcal{N} \mid L_0(s_1) = L_0(s_2)\}$

(c) $\forall s_1 \in Mask(s_2)$, either:

  (1) $\exists s'_2 \in Post_N(s_2)$ with $Post_N(s_1) \cap Mask(s'_2) = \emptyset$ and $s_1 \in Remove(s'_2)$,

  (2) $\exists s'_2 \in Post_F(s_2)$ with $Post_N(s_1) \cap Mask(s'_2) = \emptyset$ and $s_1 \notin Mask(s'_2)$ and $s_1 \in Remove(s'_2)$,

  (3) $\forall s'_2 \in Post_N(s_2) : Post_N(s_1) \cap Mask(s'_2) \neq \emptyset$.

  (4) $\forall s'_2 \in Post_F(s_2) : Post_N(s_1) \cap Mask(s'_2) \neq \emptyset \vee s_1 \in Mask(s'_2)$.

From (c), we obtain that, when $Remove(s'_2) = \emptyset$ for every $s'_2 \in S$, then: $\forall s_2 \in S : \forall s_1 \in Mask(s_2) : \forall s'_2 \in Post_F(s_2) : Post_N(s_1) \cap Mask(s'_2) \neq \emptyset \vee s_1 \in Mask(s'_2)$.

That is, the relation defined as $s_1 \prec_{Mask} s_2$ satisfies condition $B.4$, and similarly for $B.3$, of Definition 3.1.1. The proof for Algorithm 1 is similar. □

Lemma 3.5.2 (Termination of Masking)**.** *Algorithms 1 and 2 terminate.*

Proof. We prove termination of Algorithm 2; the proof for the other one is similar. The key point is to note that any state s_1 can only be inserted into $Remove(s')$ once. That is, once we process it, it will never be inspected again in line 7 of this algorithm. Note that, if $s_1 \in Remove(s'_2)$ and let s'_2 be the state that is selected in line 5, then $s_1 \notin Pre_N(Mask(s'_2))$. Moreover, since the $Mask$ sets are decreasing (line 10 and 18 of Algorithm 2), $s_1 \notin Pre_N(Mask(s'_2))$ in all further iterations. The only reason to insert s_1 in $Remove(s'_2)$ is when $s_1 \in Pre_N(s''_1)$ for some state $s''_1 \in Mask(s'_2)$ with $\{s''_1\} = Post(s_1) \cap Mask(s'_2)$. But then $s_1 \in Pre_N(Mask(s'_2))$, which is a contradiction, and therefore s_1 will never be added again to $Remove(s'_2)$. □

Theorem 3.5.3 (Complexity of Masking)**.** *The masking fault-tolerance relation $\prec_{Mask}$ of a colored Kripke structure $M = \langle S, I, R, L, \mathcal{N} \rangle$ for sublabeling $L_0 \subseteq L$ obtained by restricting AP to AP', can be computed with Algorithms 1 and 2 in a running time of $\mathcal{O}(|S|^2 * |AP'| + |E| * |S|)$ where $|E|$ is the number of edges of the structure and $|S|$ the number of states.*

Proof. Let us prove the result for Algorithm 2; the proof for the other algorithm is similar. Let $|E|$ be the number of edges of M. Similar to [Henzinger et al., 1995], we use an array to keep track of the numbers $count(s_1, s'_2) = |Post_N(s_1) \cap Mask(s'_2)|$. The initialization of $Mask(s)$ for any s can be done in time $\mathcal{O}(|S| * |AF'|)$; thus, initializing $Mask$ takes $\mathcal{O}(|S|^2 * |AF'|)$ time. On the other hand, $Remove(s)$ can be computed in time $\mathcal{O}(|S|)$ for any s. Then calculating line 3 of the algorithm takes at most $\mathcal{O}(|S|^2)$ (or $\mathcal{O}(|E|)$) steps. The loop of line 5 is executed at most once for each s_1 and $s \rightarrow s_1$; furthermore, using the array *counters*, lines $9 - 14$ can be computed in time $\mathcal{O}(|S|)$ in the worst case; thus, the entire loop takes time $\mathcal{O}(|E| * |S|)$. The same argument holds for lines $16 - 24$. Thus the algorithm has a running time of $\mathcal{O}(|S|^2 * |AP'| + |E| * |S|)$, which in the worst case is $\mathcal{O}(|S|^2 * |AP'| + |S|^3)$. □

3.5.2 Computing Nonmasking Fault-Tolerance

The basic scheme for computing a nonmasking fault-tolerant relation is sketched in Algorithm 3, which takes as input a colored Kripke structure $M = \langle S, I, R, L, \mathcal{N} \rangle$ and

a sub-labelling $L_0 \subseteq L$, and computes a nonmasking fault-tolerant $\prec_{Nonmask}$ relation that satisfies conditions $B.3$, $B.4$ and $B.5$. Note that condition $B.2$ can be checked with Algorithm 1.

Let us explain Algorithm 3. For each $s_2 \in S$, the set $NMask(s_2)$ will contain the normal states that are nonmasking for s_2. Initially, $NMask(s_2)$ consists of all normal states with the same labels as s_2 while $Remove(s_2)$ contains all the normal states which do not have successor states simulating some successor state from s_2. We also consider a set $Remove^+(s_2)$, which intuitively contains those states that do not have successor states simulating some reachable state from s_2. Both sets are updated while inspecting the structure. This is the main difference with Algorithm 2. Note that in the algorithm we use the set of all states reachable from a state s starting with a faulty action; this is defined formally as: $Post^+(s) = \{s'' \mid \exists s' \in Post_F(s) : s' \Rightarrow s''\}$. Similarly, we define the transitive-reflexive closure as: $Post^*(s) = \{s'' \mid s \Rightarrow^* s''\}$. Inside the loop of lines 6-35, we compute the sets $Remove$ and $Remove^+$ and the collection $NMask(s)$ is updated following conditions $B.3$, $B.4$ and $B.5$.

Note that the transitive closure can be computed in cubic time with respect to the set of states; however, in practice, when constructing the graph that describes our system we can also construct the transitive closure at the same time, improving the complexity of Algorithm 3. Let us prove the correctness of the algorithm for computing nonmasking relations.

Theorem 3.5.4 (Partial Correctness of Algorithm 3)**.** *On termination, Algorithm 3 returns* $\prec_{Nonmask}$.

Proof. We have to prove that Algorithm 3 ensures conditions $B.4$ and $B.5$ of Definition 3.2.1. In the first place, notice that the loop of line 5 maintains the following loop invariant. For all faulty state $s_2 \in \mathcal{F}$:

(a) $Remove(s_2) \subseteq \mathcal{N} \backslash Pre_N(Nonmask(s_2))$

(b) $Remove^+(s_2) \subseteq \mathcal{N} \backslash Pre_N(Nonmask(Post^*(s_2)))$

(c) for any relation $\prec_{Nonmask}$: $\{s_1 \in \mathcal{N} \mid s_1 \prec_{Nonmask} s_2\} \subseteq NMask(s_2) \subseteq \{s_1 \in \mathcal{N} \mid L_0(s_1) = L_0(s_2)\}$

(d) $\forall s_1 \in NMask(s_2)$, either:

Algorithm 3 Computation of nonmasking fault-tolerant

Require: colored Kripke structure M with set of states S
Ensure: Conditions $B.3$, $B.4$ and $B.5$ are checked.

1: **for all** $s_2 \in S$ **do**
2: $NMask(s_2) := \{s_1 \in \mathcal{N} \mid L_0(s_1) = L_0(s_2)\}$
3: $Remove(s_2) := \mathcal{N} \backslash Pre_N(NMask(s_2))$
4: $Remove^+(s_2) := \mathcal{N} \backslash Pre_N(NMask(Post^*(s_2)))$
5: **end for**
6: **while** $\exists\ s_2' \in S$ with $Remove(s_2') \cup Remove^+(s_2') \neq \emptyset$ **do**
7: select s_2' such that $Remove(s_2') \cup Remove^+(s_2') \neq \emptyset$
8: **for all** $s_1 \in Remove(s_2') \cup Remove^+(s_2')$ **do**
9: **if** $s_1 \in Remove(s_2')$ **then**
10: **for all** $s_2 \in Pre_N(s_2')$ **do**
11: **if** $s_1 \in NMask(s_2')$ **then**
12: $NMask(s_2) := NMask(s_2) \backslash \{s_1\}$
13: **for all** $s \in Pre_N(s_1)$ with $Post_N(s) \cap NMask(s_2) = \emptyset \wedge (s \notin NMask(Pre_F(s_2)) \vee s_1 \in RemoveF(s_2'))$ **do**
14: $Remove(s_2) := Remove(s_2) \cup \{s\}$
15: **if** $s_1 \in NMask(s_2) \wedge s_1 \in Remove^+(s_2')$ **then**
16: $Remove^+(s_2) := Remove^+(s_2) \cup \{s\}$
17: **end if**
18: **end for**
19: **end if**
20: **end for**
21: **end if**
22: **if** $s_1 \in Remove^+(s_2')$ **then**
23: **for all** $s_2 \in Pre_F(s_2')$ **do**
24: **if** $s_1 \in NMask(s_2')$ **then**
25: $NMask(s_2) := NMask(s_2) \backslash \{s_1\}$
26: **for all** $s \in Pre_N(s_1)$ with $Post_N(s) \cap NMask(Post_F^*(s_2)) = \emptyset$ **do**
27: $Remove^+(s_2) := Remove(s_2) \cup \{s\}$
28: $Remove(s_2) := Remove(s_2) \cup \{s\}$
29: **end for**
30: **end if**
31: **end for**
32: **end if**
33: **end for**
34: $Remove(s_2) := \emptyset$
35: **end while**
36: **return** $\{\langle s_1, s_2 \rangle \mid s_1 \in NMask(s_2) \wedge s_2 \in SimB2(s_1)\}$

(1) $\exists s'_2 \in Post_N(s_2)$ with $Post_N(s_1) \cap NMask(s'_2) = \emptyset \wedge s_1 \in Remove(s'_2)$, or

(2) $\exists s'_2 \in Post_F(s_2)$ with $Post_N(s_1) \cap NMask(Post^+(s'_2)) = \emptyset \wedge s_1 \in Remove^+(s'_2)$, or

(3) $\forall s'_2 \in Post_N(s_2) : Post_N(s_1) \cap Nonmask(s'_2) \neq \emptyset$ and $\forall s'_2 \in Post^+(s_2) : Post_N(s_1) \cap Nonmask(s'_2) \neq \emptyset$

From (c), we obtain that, when $Remove(s'_2) = \emptyset$ for some $s'_2 \in S$, then: $\forall s'_2 \in Post_N(s_2) : Post_N(s_1) \cap Nonmask(s'_2) \neq \emptyset$ and $\forall s'_2 \in Post^+(s_2) : Post_N(s_1) \cap Nonmask(s'_2) \neq \emptyset$, that is, conditions $B.3$, $B.4$ and $B.5$ of Definition 3.2.1 hold. □

The proof of termination of Algorithm 3 is similar to the proof of Lemma 3.5.2; the reader is referred to that theorem. With respect to the complexity of computing the nonmasking fault-tolerance relation $\prec_{Nonmask}$, it maintains polynomial complexity with respect to the traditional simulation algorithms, as it is proven in the following theorem.

Theorem 3.5.5. *The algorithm for checking nonmasking relationship is in worst case* $\mathcal{O}(|S|^4 + |S|^2 * |AP'|)$.

Proof. First, using the same reasoning as in Theorem 3.5.3, we get that verifying $B.2$ can be performed in time $O(|E| * |S|)$. Now, to calculate the set $Remove(s_2)$ and $Remove^+(s_2)$ of lines $3-4$, we need to compute the transitive closure of $\rightarrow$ which can be done in time $O(|S|^3)$. Since this set is calculated once per each state we have that lines $1-4$ take time $\mathcal{O}(|S|^4)$ and $NMask$ is calculated in $\mathcal{O}(|S|^2 * |AP'|)$. On the other hand, as explained in the proof of Theorem 3.5.3, lines $5-34$ take time $\mathcal{O}(|E| * |S|)$. Thus in the worst case the time complexity of the algorithm is $\mathcal{O}(|S|^4 + |S|^2 * |AP'|)$. □

3.5.3 Computing Failsafe Fault-Tolerance

We finally present an algorithm to calculate a relation of failsafe fault-tolerance between two systems, in the case it exists. The scheme of this algorithm is similar to that of Algorithm 2. This is because both masking and failsafe tolerance require that the safety properties have to be guaranteed. However, liveness properties are not necessarily preserved in failsafe tolerance. The main difference with Algorithm 2 is

in lines 12 and 18, where we allow the faulty system to stay in a safe set of sates. To this end, we introduce a set $Eq(s)$ containing the closure of reachable states with the same labeling. In the case that there exists some reachable state from the origin with a different labeling, we set $Eq(s)$ to empty.

Summarizing, in Algorithm 4, for each $s_2 \in S$, the set $FSafe(s_2)$ contains those normal states that are candidates for simulating s_2. Initially, $FSafe(s_2)$ consists of all normal states with the same labels as s_2; the set $Remove(s_2)$ is similar to its characterization in Algorithm 2. Note that in lines 12 and 18, in the case that the simulation relation is not preserved, the system has the option to stay in a set of safe states.

The proofs of correctness and termination are similar to Theorem 3.5.1 and Lemma 3.5.2, with some minor changes.

Theorem 3.5.6 (Partial Correctness of Failsafe)**.** *On termination, Algorithms 4 and 1 return* $\prec_{Failsafe}$.

Proof. We have to prove that Algorithm 4 ensures conditions $B.3$ and $B.4$ of Definition 3.3.1. As explained above, condition $B.2$ is computed using Algorithm 1, which is correct by Theorem 3.5.1. For the other conditions, first note that the loop of line 5 has the following loop invariant. For all states $s_2 \in S$:

(a) $Remove(s_2) \subseteq (\mathcal{N} \backslash Pre_N(FSafe(s_2)))$

(b) for any relation $\prec_{failsafe}$: $\{s_1 \in \mathcal{N} \mid s_1 \prec_{failsafe} s_2\} \subseteq FSafe(s_2) \subseteq \{s_1 \in \mathcal{N} \mid L_0(s_1) = L_0(s_2)\}$

(c) $\forall s_1 \in FSafe(s_2)$, either:

  (1) $\exists s_2' \in Post_N(s_2)$ with $Post_N(s_1) \cap FSafe(s_2') = \emptyset$ and $s_1 \in Remove(s_2')$,

  (2) $\exists s_2' \in Post_F(s_2)$ with $Post_N(s_1) \cap FSafe(s_2') = \emptyset \wedge L_0(s_1) \neq L_0(s_2')$ and $s_1 \in Remove(s_2')$,

  (3) $\forall s_2' \in Post(s_2) : Post_N(s_1) \cap FSafe(s_2') \neq \emptyset \vee s_1 \in FSafe(s_2')$

From (c), we obtain that, when $Remove(s_2') = \emptyset$ for every $s_2' \in S$, then: $\forall s_2 \in S : \forall s_1 \in FSafe(s_2) : \forall s_2' \in Post_F(s_2) : Post_N(s_1) \cap FSafe(s_2') \neq \emptyset \vee s_1 \in FSafe(s_2')$ and also $\forall s_2 \in S : \forall s_1 \in FSafe(s_2) : \forall s_2' \in Post_N(s_2) : Post_N(s_1) \cap FSafe(s_2') \neq$

Algorithm 4 Computation of failsafe fault-tolerant

Require: colored Kripke structure M with set of states S
Ensure: Failsafe fault-tolerance relation $\prec_{Failsafe}$ for M

for all $s_2 \in S$ **do**
  $FSafe(s_2) := \{s_1 \in \mathcal{N} \mid L_0(s_1) = L_0(s_2)\}$
  $Remove(s_2) := \mathcal{N} \backslash Pre_N(FSafe(s_2))$
  $Eq(s_2) := \{s \mid s_2 \Rightarrow^* s \wedge L_0(s_2) = L_0(s) \wedge (\nexists s' : s_2 \Rightarrow^* s' \wedge L_0(s) \neq L_0(s'))\}$
end for
while $\exists\ s_2' \in S$ with $Remove(s_2') \neq \emptyset$ **do**
  select s_2' such that $Remove(s_2') \neq \emptyset$
  for all $s_1 \in Remove(s_2')$ **do**
    for all $s_2 \in Pre_N(s_2')$ **do**
      if $s_1 \in FSafe(s_2)$ **then**
        $FSafe(s_2) := FSafe(s_2) \backslash \{s_1\}$
        for all $s \in Pre_N(s_1)$ with $Post_N(s) \cap FSafe(s_2) = \emptyset \wedge (s \notin FSafe(Pre_F(s_2)) \vee FSafe(Eq(Pre_F(s_2))) \neq \emptyset)$ **do**
          $Remove(s_2) := Remove(s_2) \cup \{s\}$
        end for
      end if
    end for
    for all $s_2 \in Pre_F(s_2')$ **do**
      if $s_1 \in FSafe(s_2) \wedge Eq(s_2) = \emptyset$ **then**
        $FSafe(s_2) := FSafe(s_2) \backslash \{s_1\}$
        for all $s \in Pre_N(s_1)$ with $Post_N(s) \cap FSafe(s_2) = \emptyset$ **do**
          $Remove(s_2) := Remove(s_2) \cup \{s\}$
        end for
      end if
    end for
  end for
  $Remove(s_2) := \emptyset$
end while
return $\{\langle s_1, s_2 \rangle \mid s_1 \in FSafe(s_2) \wedge s_2 \in SimB2(s_1)\}$

$\emptyset$. That is, the relation defined as $s_1 \prec_{Mask} s_2$ satisfies conditions $B.4$ and $B.3$, respectively, of Definition 3.1.1. □

With respect to the complexity of the failsafe fault-tolerance $\prec_{Failsafe}$, a similar proof as for Algorithm 2 shows that it also has a polynomial complexity.

Theorem 3.5.7. *The time complexity of Algorithm 4 is in* $\mathcal{O}(|E| * |S| + |E| * |AP'|)$.

3.6 Some Examples

In this section we present four case studies of typical fault-tolerance situations to show how our approach can be used in practice to verify fault-tolerant systems.

3.6.1 The Muller C-element

The Muller C-element [Milner, 1980] is a simple delay-insensitive circuit which contains two boolean inputs and one boolean output. Its logical behavior is described as follows: if both inputs are true (resp. false) then the output of the C-element becomes true (resp. false). If the inputs do not change, the output remains the same. In [Arora and Gouda, 1993], the following (informal) specification of the C-element with inputs x and y and output z is given:

> *(i) Input x (resp. y) changes only if $x \equiv z$ (resp., $y \equiv z$), (ii) Output z becomes true only if $x \wedge y$ holds, and becomes false only if $\neg x \wedge \neg y$ holds; (iii) Starting from a state where $x \wedge y$, eventually a state is reached where z is set to the same value that both x and y have. Ideally, both x and y change simultaneously. Faults may delay changing either x or y.*

We consider an implementation of the C-element with a majority voting circuit involving three inputs, where an extra input u in the circuit is added. Then, the predicate $maj(x, y, u)$ returns the value of the majority circuit, which is assumed to work correctly, and is defined as $maj(x, y, u) = (x \wedge y) \vee (x \wedge u) \vee (y \wedge u)$. In addition to the traditional logical behavior of the C-element, u and z have to change at the same time, where the output z is fed back to the input u.

Figure 3.7 shows two models of this circuit. M exhibits the ideal behavior of the C-element containing only normal transitions. M' takes into account the possibility

Normal Actions:

$$x = z \wedge y = z \rightarrow x, y := \neg x, \neg y$$
$$z \neq maj(x, y, u) \rightarrow z := maj(x, y, u)$$
$$u \neq maj(x, y, u) \rightarrow u := maj(x, y, u)$$

Figure 3.5: The Muller C-element program with majority voting (fault-intolerant version).

Normal and Recovery Actions:

$$x = z \wedge y = z \rightarrow x, y := \neg x, \neg y$$
$$z \neq maj(x, y, u) \rightarrow z := maj(x, y, u)$$
$$u \neq maj(x, y, u) \rightarrow u := maj(x, y, u)$$
$$x = z \wedge y \neq z \rightarrow x := \neg x$$
$$x \neq z \wedge y = z \rightarrow y := \neg y$$

Faulty Actions:

$$x = z \wedge y = z \rightarrow x := \neg x$$
$$x = z \wedge y = z \rightarrow y := \neg y$$
$$u = z \rightarrow z := \neg u$$

Figure 3.6: The Muller C-element fault-tolerant program with majority.

of faults occurring, and provides a reaction to these. Every state in these models is composed of boolean variables x, y, u, and z, where x, y, and u represent the inputs, and z represents the output. For instance, the state s_0 contains the information $000\backslash 0$ interpreted (reading from left to right) as $x = 0$, $y = 0$, $u = 0$, and $z = 0$. Transitions are labeled by subsets of the set $\{cx, cy, cu, cz\}$ of actions; action cx (resp., cy and cu) is the action that changes input x (resp., y and u); cz is the action of changing output z. When the actions cx and cy are executed in the same transition, we just write cxy. We consider two types of faults: *(i)* a delay may occur in the arrival of some of the inputs x or y (i.e., they do not change simultaneously), and *(ii)* a delay in the signal from z to u occurs. We can observe these classes of faults in the faulty states (indicated by dashed circles) when either x and y or u and z do not match one another. Notice that these two models described in Figure 3.7 correspond to the fault-intolerant and the fault-tolerant program in Figures 3.5 and 3.6 in a guarded command style, respectively.

The relation $R_{c-element} = \{\langle s_0, t_0\rangle, \langle s_1, t_1\rangle, \langle s_2, t_2\rangle, \langle s_3, t_3\rangle\}$ is a nonmasking fault-tolerant relation for $\langle M, M'\rangle$ and the sublabelings obtained by restricting the original labelings to letters u, x, y, z. Therefore, when the majority circuit behaves correctly,

this implementation masks delays of inputs.

3.6.2 The Byzantine Generals Problem

An interesting example of a fault-tolerant system is the Byzantine generals problem, introduced originally in [Lamport et al., 1982]. This is an agreement problem, where we have a general with $n-1$ lieutenants. The communication between the general and his lieutenants is performed through messengers. The general may decide to attack an enemy city or to retreat; then, he sends the order to his lieutenants. Some of the lieutenants might be traitors. As a consequence, traitors might deliver false messages or perhaps they avoid sending a message that they received. The loyal lieutenants must agree on attacking or retreating after $m+1$ rounds of communication, where m is the maximum numbers of traitors. The algorithm can ensure correct operation only if fewer than one third of the lieutenants are traitors. We assume the following: $\mathsf{L_0}$ is the general, the messages are delivered correctly and all the lieutenants can communicate directly with each other; in this scenario they can recognize who is sending a message. Faults can convert loyal lieutenants into traitors. Finally, traitors cannot forge messages on behalf of loyal lieutenants.

In Figure 3.8 two models of this problem are described. M exhibits the ideal behavior of the Byzantine agreement for four loyal lieutenants $\mathsf{L_0}$, $\mathsf{L_1}$, $\mathsf{L_2}$, and $\mathsf{L_3}$. On the other hand, M' expresses the same behavior that M does, but considering the presence of lieutenant $\mathsf{L_1}$ as a traitor. We specify this problem following the ideas introduced in [Castro and Maibaum, 2009b]. We have the following propositions: $\mathsf{L_i.A_{l_1,\ldots,l_n}}$ (this proposition indicates that $\mathsf{L_i}$ has received a message from lieutenants $\mathsf{l_1,\ldots,l_n}$ saying that he must attack). We have a violation proposition $\mathsf{L_i.traitor}$ for each lieutenant (this proposition is true when $\mathsf{L_i}$ is a traitor) and $\mathsf{L_i.d}$ (this proposition is true when $\mathsf{L_i}$ has decided to attack), $\mathsf{r_i}$ (this proposition is true when we are in round i). Each state has the propositions ($\mathsf{P_1}$, $\mathsf{P_2}$, $\mathsf{P_3}$, or $\mathsf{P_4}$) which hold in it and the relevant information about the current round. Each transition is labeled with some of the following actions: $\mathsf{L_i.sendA(L_{l_1,\ldots,l_n})}$ (lieutenant $\mathsf{L_i}$ sends to lieutenants $\mathsf{l_1,\ldots,l_n}$ the message of attack), $\mathsf{L_i.fwd(L_k, A, L_{l_1,\ldots,l_n})}$ (the lieutenant $\mathsf{L_i}$ forwards to lieutenants $\mathsf{l_1,\ldots,l_n}$ the message of attack that he received from $\mathsf{L_k}$), $\mathsf{L_i.betray}$ (lieutenant $\mathsf{L_i}$ becomes a traitor). We consider a clock that allows lieutenants to synchronize; the action tt increments the clock by one unit of time. The specification uses $\mathsf{m}+1$

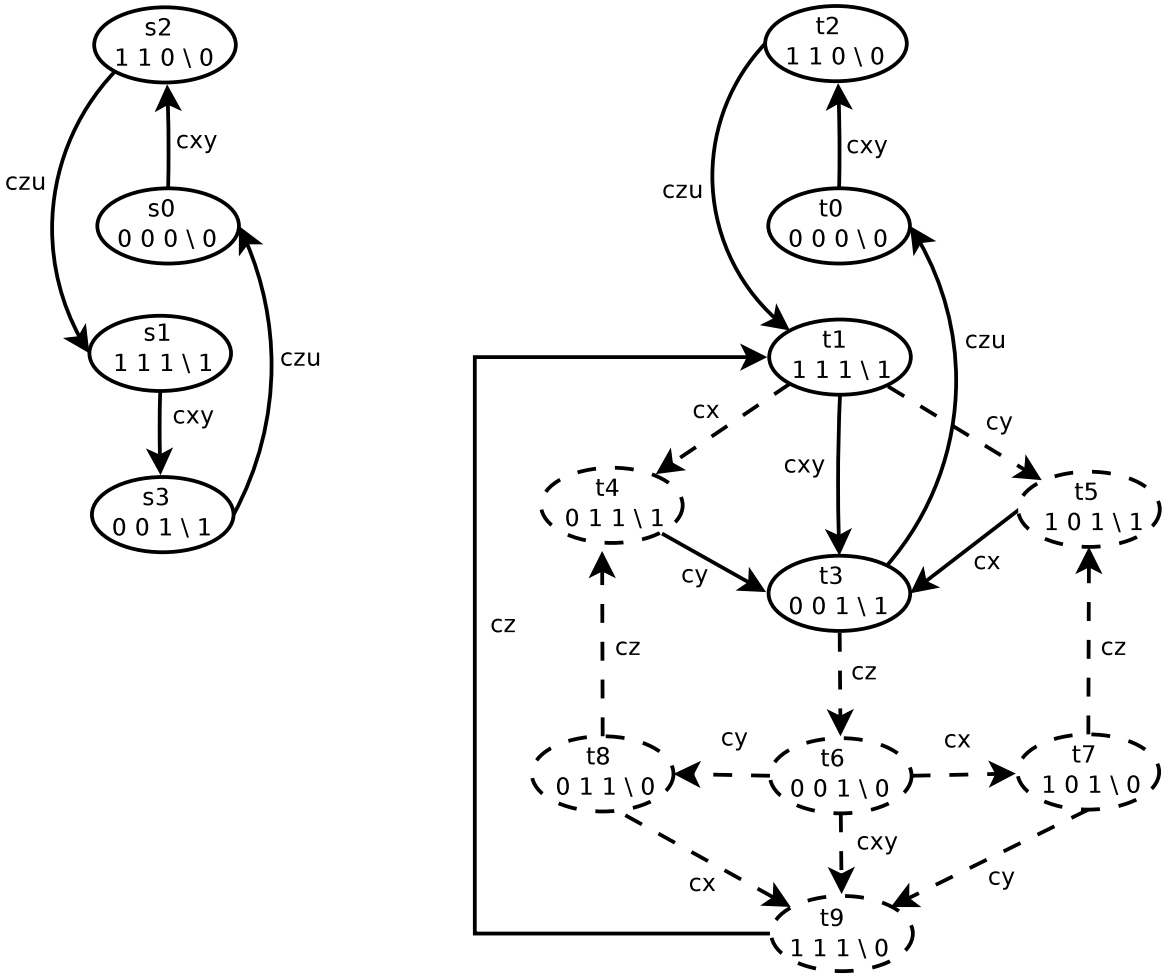

Figure 3.7: A nonmasking fault-tolerance for the Muller C-element with a majority circuit.

rounds of messages, which are coordinated by means of the clock, where m is the number of traitors for which the specification ensures that the loyal lieutenants will agree on a decision.

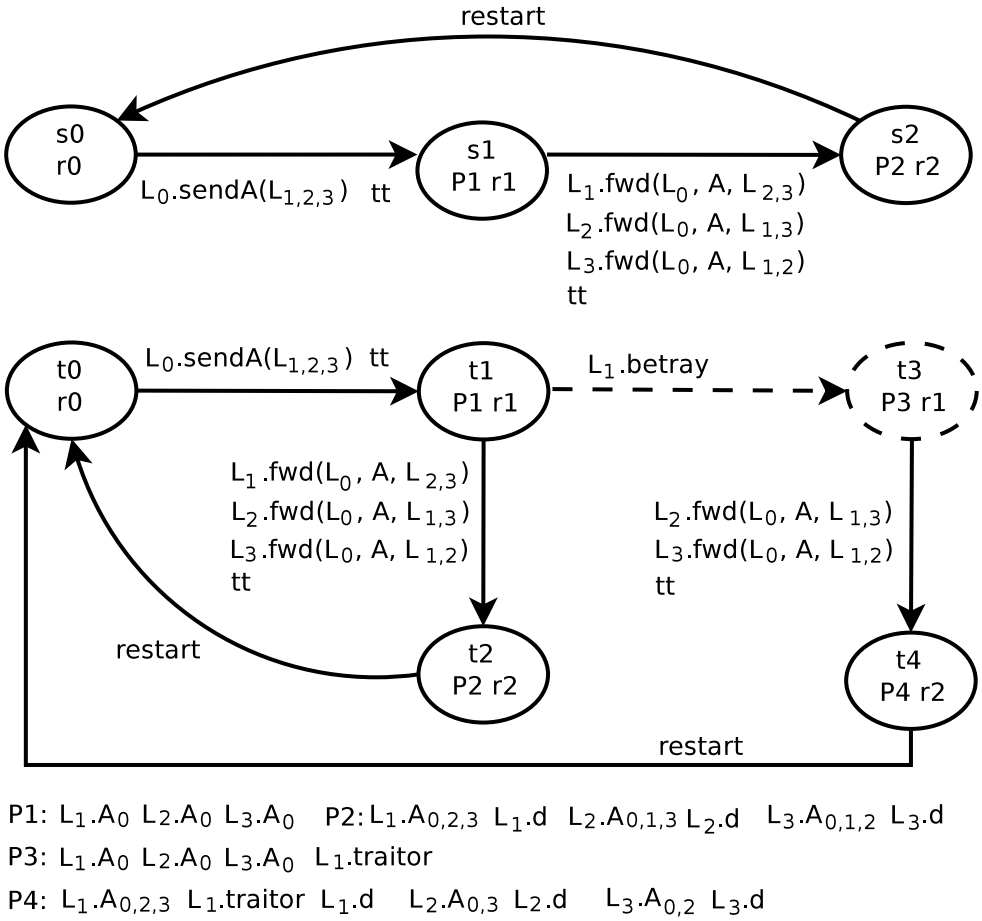

Figure 3.8: A masking fault-tolerance for the Byzantine generals problem.

In this case the relation

$$R_{byzantines} = \{\langle s_0, t_0\rangle, \langle s_1, t_1\rangle, \langle s_2, t_2\rangle, \langle s_1, t_3\rangle, \langle s_2, t_4\rangle\}$$

is a masking fault-tolerant relation for (M, M') where the sub-labelings are obtained by restricting the original labeling to $AP \setminus \{\mathsf{L_0.traitor}, ..., \mathsf{L_n.traitor}\}$; this means that the model on the bottom tolerates the existence of one traitor. This can be generalized

to support more lieutenants and traitors.

3.6.3 Altitude Switch (ASW)

The Altitude Switch (ASW) controller in an avionics system is responsible for turning on a Device of Interest (DOI) when the aircraft altitude is below a pre-specified threshold. We have adapted this real-world avionics example from [Jeffords et al., 2009].

Basically, the ASW controller reads a set of variables and produces an output. There exist four internal variables, a mode variable that determines the operating mode of the system, and four input variables that represent the state of the altitude sensors. The internal variables are as follows: (1) *AltBelow* is true if the altitude is below a pre-specified threshold. (2) *DOIStatus* is true if the DOI is powered on; (3) *Inhibit* is true when the DOI power-on is inhibited, and (4) *Reset* is true if the system is being reset. The ASW program can be in three different modes: (1) the *Initialization* (*Init*) mode when the ASW system is initializing; (2) the *AwaitDOIon* (*AD*) mode if the system is waiting for the DOI to power on, and (3) the *Standby* (*SB*) mode for all other cases.

The ASW system can be subject to hardware malfunctions that may alter the behavior of the ASW controller. In order to deal with potential faults, the system is designed to tolerate three time-out faults: (1) initialization fault ($InitFail$), (2) altimeter fault ($AltFail$), and (3) DOI fault ($DOIFail$). These types of faults require the system to stay in a given state for a specific amount of time. Because we do not include the notion of time in this example, we model these faults as on/off flags. As a consequence of these malfunctions, a new mode, *Fault* (FM), is added to the mode class to indicate the presence of faults in the system.

Figure 3.9 shows two models of this program. M exhibits the ideal behavior of the ASW controller containing only normal transitions. M' takes into account the possibility of faults occurring, and provides a reaction to these. Every state in these models identifies some of the modes: $Init$, AD, SB, or FM. Transitions are labeled by actions that represent the changes among the different modes. Action *compInit* moves the system from $Init$ to SB once the initialization is complete, requiring this as a precondition. It returns to $Init$ when the pilot pushes the *reset* button. Moreover, the system moves from SB to AD through the action *altBelow* when the aircraft

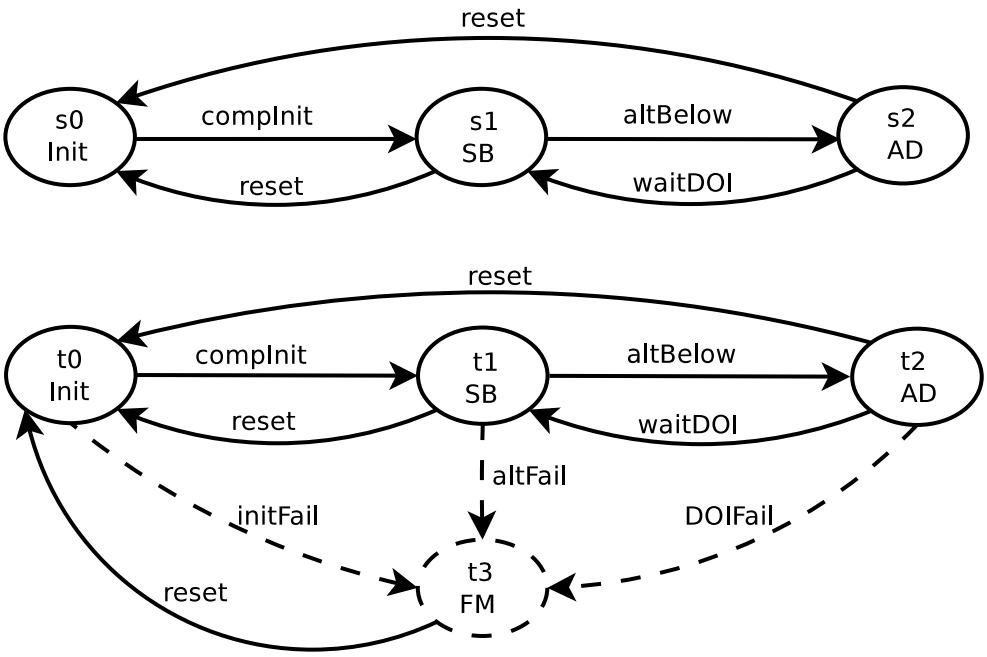

Figure 3.9: A nonmasking fault-tolerance for the Altitude Switch Controller.

descends below the threshold altitude, but requiring as precondition that powering on is not inhibited, and the DOI is not powered on. Once the DOI signals that it is powered on, the system goes from AD to SB. Furthermore, the system returns from AD to $Init$ when the pilot pushes the $reset$ button. All these actions correspond to the ideal behavior of the ASW program. On the other hand, actions $initFail$, $altFail$, and $DOIFail$ model the occurrences of faults, perturbing the normal behavior of the system. Consequently, when the system *detects* any of these faults, it goes into the faulty mode (FM). Finally, the problem specification requires that the program does not change its mode from *Standby* to $AwaitDOI$ if the altitude sensors failed, i.e., when $AltFail$ is true. Moreover, from the faulty state FM, the program can only go into the $Initialization$ mode. In fact, the program can recover from the *faulty* state if the system has been reset by the pilot.

As a result, the relation $R_{asw-controller} = \{\langle s_0, t_0\rangle, \langle s_1, t_1\rangle, \langle s_2, t_2\rangle, \langle s_0, t_3\rangle\}$ is a nonmasking fault-tolerant relation for $\langle M, M'\rangle$. We obtain a similar result compared to [Jeffords et al., 2009], where the authors call eventual masking ("*the system enters a fault handling state but eventually recovers to normal behavior*") for this kind of fault-tolerant behavior.

3.6.4 A Simple Train System

We now consider a simple train system. Train systems control the movement of trains through a network of rail segments. Fault-tolerance is a key aspect of these systems: a fault in the system may cause a train collision and the loss of human life. These kinds of systems are the object of active research in the fault-tolerance community (see [Abdelouahab and Braga, 2008; Abrial, 2006; Guiho and Hennebert, 1990]).

The general version of our system consists of n trains and m rail segments. Rail segments are connected to other rail segments, where in each of these connections the rails are equipped with a signal which indicates if the segment is occupied or not. The signals can be green (when the segment is free) or red (when another train is in the segment). We have the following predicates. For each $0 \leq i \leq n$ and $0 \leq j \leq m$, we have a predicate $t_i.r_j$ which indicates that train i is in the segment j; the predicate $r_j.green$ ($r_j.red$) expresses that the signal of segment j is green (red), respectively. Moreover, $t_i.stop$ denotes that train t_i is stopped and r_iRr_j means that segments i and j are connected. We have the following actions: $t_i.move(j)$: train t_i moves to segment j, $t_i.stops$: train t_i stops, $r_i.ggreen$: the signal of r_i is set to green, and $r_i.gred$: the signal of r_i is set to red.

The specification requires that, if there is a train in a segment, then the signal for this segment must be red. On the contrary, if there is no train in the segment, then the signal for the segment must be green. Moreover, when a train detects that another train is already in the same segment, a fault has occurred, and it ought to stop to avoid train collisions. Besides, when a train is in a segment where all the connected segments have their signals set to red, the train will stop. Finally, an ideal safety property of this system is that there are not two trains in the same segment; that is, this property must hold in the specification when no faults are observed.

This system is implemented by a sensor in each segment which detects the existence of two trains. Furthermore, the movement of trains between rail segments is controlled by a pair of sensors called *Block Instruments* in railways. Essentially, these devices have the task of activating the green or red signal. The basis of operation of the system is very simple, but what is more significant is the inherent fail-safety built into it. In case a fault occurs, the interlocking relay logic turns the signal to red.

In order to provide a better understanding of this example and the particular situation for failsafe fault-tolerance, let us consider trains t_i with $0 \leq i \leq 3$ and

rail segments r_j with $0 \leq j \leq 4$, where they are connected as follow: r_0Rr_1, r_1Rr_2, r_2Rr_3, r_2Rr_4, and r_3Rr_4. In Figure 3.10 two models of this problem are described. M exhibits the ideal behavior of the simple train system for three trains on five rail segments. On the other hand, M' expresses the same behavior that M does, but considering the presence of faults. Notice that in this figure is illustrated part of the behavior for this set of train and rail segments, starting from a state s_0 in which trains are already located on the rail segments. The states are labeled with predicates (P_0, P_1, P_2, or P_3) pointing out the formulas holding in each state. Moreover, transitions are labeled with some of the actions described above.

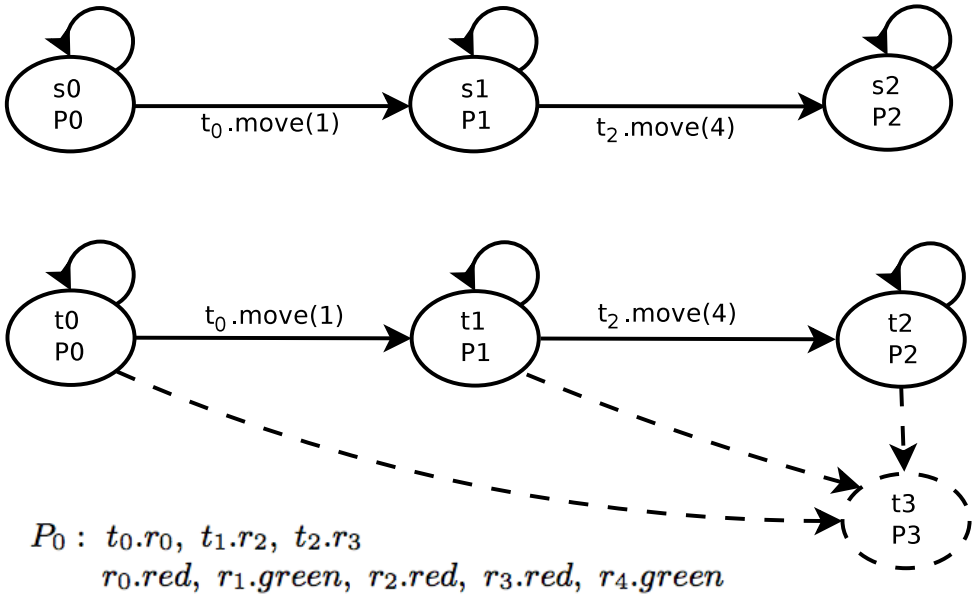

Figure 3.10: A failsafe fault-tolerance for a Simple Train System.

Regarding the model M', faults may occur that affect the communication of the system or the behavior of the sensors. The block instruments on each rail segment react to this malfunction, turning the signal to red (P_3). It is clear that this state still

guarantees the (safety) property that no two trains are located in the same segment. However, reaching state t_3 means that all trains will stop and they will stay there until an external recovery action is performed. In this case the relation

$$R_{train-system} = \{\langle s_0, t_0\rangle, \langle s_1, t_1\rangle, \langle s_2, t_2\rangle\}$$

is a failsafe fault-tolerant relation for $\langle M, M'\rangle$.

Notice that a fault leading to a failsafe state is not considered as a failure in a safety critical system. The main idea of a failsafe design is to detect the faults and mask its impact until some recovery actions are under taken.

Chapter 4

The Synthesis Approach

In this chapter, we address the second core problem of the dissertation: our synthesis method. We present the extension of a synthesis algorithm for CTL to cope with dCTL- specifications. Moreover, we explain the details of each of the algorithms for the three degrees of fault-tolerance, as well as analysing their complexity. Finally, we prove some properties of our method, like soundness and completeness.

4.1 The Synthesis Problem

The problem of synthesis of fault-tolerant programs is as follows, where we require the following inputs:

(1) a deontic specification *dSpec*: *init-spec* and *normal-spec*, where *init-spec* specifies the initial state, and *normal-spec* specifies correctness properties using dCTL- formulas that are required to hold at all states that are reachable from an initial state in the absence of faults,

(2) a fault specification $fSpec$: *fault-variables*, *fault-spec*, and *combine-spec*, where *fault-variables* is a set of auxiliary atomic propositions, and *fault-spec* specifies the faulty and recovery behavior over the atomic propositions (including *fault-variables*), using CTL formulas. Finally, *combine-spec* are also CTL formulas which relate the atomic propositions in the deontic specification *dSpec* with those in the fault specification $fSpec$,

(3) a desired level of fault-tolerance (masking, nonmasking or failsafe) chosen by the user, and

(4) an *interface* represented by a subset of the state variables, which intuitively constitutes the visible part of the system.

The aim is to synthesize a fault-tolerant component that:

(1) in the absence of faults, satisfies the deontic specification $dSpec$, and

(2) in the presence of faults, satisfies the required level of fault-tolerance based on the *interface* and the fault specification $fSpec$.

In other words, our goal is to automatically determine whether a fault-tolerant component, with the required level of fault-tolerance, is realizable. Moreover, if the answer is positive, the goal is to algorithmically produce such a fault-tolerant implementation.

Note that the deontic system specification $dSpec$ involves the use of CTL to describe the system declaratively (including safety and liveness properties of the system), while the deontic operators of dCTL- allow us to capture *obligations*, and to indirectly characterize faults as events violating these obligations. Notice that the deontic specification states what the expected behavior of the system is, and, indirectly, what the possible faults are. In other words, the possible faults may not be explicitly given by the user, as in other approaches, but stated at the specification level. However, we allow the user to provide their own fault specification $fSpec$ which is similar to the one required in [Attie et al., 2004]. In our case the faulty behavior is specified using CTL formulas instead of guarded commands. We present several examples in Chapter 5.

Roughly speaking, the synthesis process is based on the extraction of a finite behavior model from a dCTL- specification. This is achieved by constructing a behavior model that captures the system augmented with faults, and then combining a synthesis algorithm for dCTL- with simulation relations that capture masking, nonmasking, or failsafe fault-tolerance, in order to remove from this model those states and faults that lie outside the required level of tolerance. The synthesis algorithm aims at detecting the *maximal* set of faults that can be tolerated (for the required

level of fault-tolerance), and returning a (maximal) program that provides recovery from these faults.

4.2 The dCTL- Decision Procedure

In this section we provide all the details of the dCTL- decision procedure. As we mention, our work is strongly related to the one of Attie et al. [Attie et al., 2004]. Essentially, we present here the extension of the decision procedure for CTL to cope with dCTL- specifications. Fundamentally, we want to use this procedure to check whether the input deontic system specification *dSpec* is satisfiable in the absence of faults.

The decision procedure takes as input a formula f_0 and returns as a result *true* if f_0 is satisfiable or $false$ in the case f_0 is unsatisfiable. If f_0 is satisfiable, a finite model is constructed. The decision procedure performs the following steps:

(1) Construct the initial tableau T_0 which encodes potential models of f_0. If f_0 is satisfiable, then a final model is embedded in T_0.

(2) Check the consistency of the tableau by removing inconsistent part of it. As a result, if the "root" of the tableau is deleted, f_0 is unsatisfiable. Otherwise, f_0 is satisfiable.

(3) Finally, a model M_N of f_0 is built by unraveling the tableau T_0.

The decision procedure begins by building an AND/OR graph (tableau), formally defined as follow:

Definition 4.2.1. (AND/ OR Graph). An AND/OR graph (tableau) K is a tuple $(V_C, V_D, A_{CD}, A_{DC}, L)$ with the following components:

(1) V_C, a set of AND-nodes,

(2) V_D, a set of OR-nodes,

(3) $A_{CD} \subseteq V_C \times V_D$, a set of AND-OR transitions,

(4) $A_{DC} \subseteq V_D \times V_C$, a set of OR-AND transitions,

(5) $L : V_C \cup V_D \rightarrow 2^{cl(f)}$ is an interpretation function, which denotes the set of formulas (subset of Fisher-Ladner Closure $cl(f)$, defined below) that holds in each node in $V_C \cup V_D$.

We note that this definition is reproduced and slightly modified from [Attie et al., 2004], the difference being that we do not consider process indexes indicating which nondeterministic choice is made in a concurrent program.

Each node of K is either an OR-node or an AND-node and is labelled by a set of formulas. The tableau K has a root node $d_0 = \{f_0\}$ from which all other nodes in T are reachable. We use c, c', ... to denote AND-nodes, d, d', ... to denote OR-nodes, and e, e', ... to denote nodes of either type. Each node is labeled with a subset of $cl(f)$ (see below), and no two AND-nodes nor two OR-nodes are equally labeled.

Definition 4.2.2. (Fisher-Ladner Closure). If f is a dCTL- formula, then $cl(f)$, the generalized Fisher-Ladner closure of f, is defined as follows:

$cl(p) = p$ for atomic proposition p,

$cl(\phi \wedge \psi) = \{\phi \wedge \psi\} \cup cl(\phi) \cup cl(\psi)$,

$cl(\neg\phi) = \{\neg\phi\} \cup cl(\phi)$,

$cl(\mathsf{A}(\phi \mathcal{W} \psi)) = \{\mathsf{A}(\phi \mathcal{W} \psi), \mathsf{AXA}(\phi \mathcal{W} \psi)\} \cup cl(\phi) \cup cl(\psi)$,

$cl(\mathsf{E}(\phi \mathcal{W} \psi)) = \{\mathsf{E}(\phi \mathcal{W} \psi), \mathsf{EXE}(\phi \mathcal{W} \psi)\} \cup cl(\phi) \cup cl(\psi)$

$cl(\mathsf{AG}\ \phi) = \{\mathsf{AG}\ \phi, \mathsf{AXAG}\ \phi\} \cup cl(\phi)$

$cl(\mathsf{EG}\ \phi) = \{\mathsf{EG}\ \phi, \mathsf{EXEG}\ \phi\} \cup cl(\phi)$

$cl(\mathsf{AX}\ \phi) = \{\mathsf{AX}\ \phi\} \cup cl(\phi)$

$cl(\mathsf{EX}\ \phi) = \{\mathsf{EX}\ \phi\} \cup cl(\phi)$

$cl(\mathsf{A}(\phi \mathcal{U} \psi)) = \{\mathsf{A}(\phi \mathcal{U} \psi), \mathsf{AXA}(\phi \mathcal{U} \psi)\} \cup cl(\phi) \cup cl(\psi)$

$cl(\mathsf{E}(\phi \mathcal{U} \psi)) = \{\mathsf{E}(\phi \mathcal{U} \psi), \mathsf{EXE}(\phi \mathcal{U} \psi)\} \cup cl(\phi) \cup cl(\psi)$

$cl(\mathsf{AF}\ \phi) = \{\mathsf{AF}\ \phi, \mathsf{AXAF}\ \phi\} \cup cl(\phi)$

$$cl(\mathsf{EF}\ \phi) = \{\mathsf{EF}\ \phi, \mathsf{EXEF}\ \phi\} \cup cl(\phi)$$

$$cl(\mathbf{O}(\phi\,\mathcal{U}\,\psi)) = \{\mathbf{O}(\phi\,\mathcal{U}\,\psi), \mathsf{AXO}(\phi\,\mathcal{U}\,\psi)\} \cup cl(\phi) \cup cl(\psi)$$

$$cl(\mathbf{P}(\phi\,\mathcal{U}\,\psi)) = \{\mathbf{P}(\phi\,\mathcal{U}\,\psi), \mathsf{EXP}(\phi\,\mathcal{U}\,\psi)\} \cup cl(\phi) \cup cl(\psi)$$

$$cl(\mathbf{O}(\phi\,\mathcal{W}\,\psi)) = \{\mathbf{O}(\phi\,\mathcal{W}\,\psi), \mathsf{AXO}(\phi\,\mathcal{W}\,\psi)\} \cup cl(\phi) \cup cl(\psi)$$

$$cl(\mathbf{P}(\phi\,\mathcal{W}\,\psi)) = \{\mathbf{P}(\phi\,\mathcal{W}\,\psi), \mathsf{EXP}(\phi\,\mathcal{W}\,\psi)\} \cup cl(\phi) \cup cl(\psi)$$

Part of this definition is reproduced from [Attie et al., 2004] and extended by including the deontic operators.

In the following subsections we describe the dCTL- decision procedure in more detail.

4.2.1 Building the initial AND/OR graph

We start constructing the initial AND/OR graph T in stages following the main idea from [Attie et al., 2004] and [Clarke and Emerson, 1981], by the method below:

(1) Initially, let the root node of T be an OR-node d_0 labeled with f_0 ($L(d_0) = \{f_0\}$).

(2) If the set of frontier nodes (those nodes without successors) of T is empty, then halt. Otherwise, let e be any frontier node in T. If e is an OR-node d, then generate the successor AND-nodes $c_1, ..., c_k$ from d and add a transition from d to c_i in T. If any AND-node c_i has the same label as another AND-node c already present in T, then merge c_i and c. If e is an AND-node c, then generate the successor OR-nodes $d_1, ..., d_k$ from c and add a transition from c to d_i in T. If any OR-node d_j has the same label as some other OR-node d already present in T, then merge d_j and d. Repeat this step.

4.2.2 Successors of OR-nodes

A dCTL- formula is *elementary* iff it is an atomic proposition, the negation of an atomic proposition, or has either AX or EX as its main connective. Nonelementary formulas are classified as either a conjunctive formula $\alpha \equiv \alpha_1 \wedge \alpha_2$ or a disjunctive formula $\beta \equiv \beta_1 \vee \beta_2$. In [Attie et al., 2004], CTL formulas are classified as follows:

$\alpha = \phi \wedge \psi$	$\alpha_1 = \phi$	$\alpha_2 = \psi$
$\alpha = \mathsf{A}(\phi \,\mathcal{W}\, \psi)$	$\alpha_1 = \psi$	$\alpha_2 = \phi \vee \mathsf{AXA}(\phi \,\mathcal{W}\, \psi)$
$\alpha = \mathsf{E}(\phi \,\mathcal{W}\, \psi)$	$\alpha_1 = \psi$	$\alpha_2 = \phi \vee \mathsf{EXE}(\phi \,\mathcal{W}\, \psi)$
$\alpha = \mathsf{AG}\ \phi$	$\alpha_1 = \phi$	$\alpha_2 = \mathsf{AXAG}\ \phi$
$\alpha = \mathsf{EG}\ \phi$	$\alpha_1 = \phi$	$\alpha_2 = \mathsf{EXEG}\ \phi$

$\beta = \phi \vee \psi$	$\beta_1 = \phi$	$\beta_2 = \psi$
$\beta = \mathsf{A}(\phi \,\mathcal{U}\, \psi)$	$\beta_1 = \psi$	$\beta_2 = \phi \wedge \mathsf{AXA}(\phi \,\mathcal{U}\, \psi)$
$\beta = \mathsf{E}(\phi \,\mathcal{U}\, \psi)$	$\beta_1 = \psi$	$\beta_2 = \phi \wedge \mathsf{EXE}(\phi \,\mathcal{U}\, \psi)$
$\beta = \mathsf{AF}\ \phi$	$\beta_1 = \phi$	$\beta_2 = \mathsf{AXAF}\ \phi$
$\beta = \mathsf{EF}\ \phi$	$\beta_1 = \phi$	$\beta_2 = \mathsf{EXEF}\ \phi$

Furthermore, the α and β classification of CTL formulas given above for tableau can be extended to our setting. For the deontic operators we proceed as follows:

$\beta = \mathbf{O}(\phi \,\mathcal{U}\, \psi)$	$\beta_1 = O_\psi$	$\beta_2 = O_\phi \wedge \mathsf{AX}\mathbf{O}(\phi \,\mathcal{U}\, \psi)$
$\beta = \mathbf{P}(\phi \,\mathcal{U}\, \psi)$	$\beta_1 = O_\psi$	$\beta_2 = O_\phi \wedge \mathsf{EX}\mathbf{P}(\phi \,\mathcal{U}\, \psi)$
$\alpha = \mathbf{O}(\phi \,\mathcal{W}\, \psi)$	$\alpha_1 = O_\psi$	$\alpha_2 = O_\phi \vee \mathsf{AX}\mathbf{O}(\phi \,\mathcal{W}\, \psi)$
$\alpha = \mathbf{P}(\phi \,\mathcal{W}\, \psi)$	$\alpha_1 = O_\psi$	$\alpha_2 = O_\phi \vee \mathsf{EX}\mathbf{P}(\phi \,\mathcal{W}\, \psi)$

where O_ϕ is obtained by substituting in ϕ any propositional variable p by a fresh variable denoted by $Opr\ p$, and similarly for O_ψ.

Note that nonelementary formulas whose main connective is a temporal modality are classified according to the fixpoint characterization of the modality.

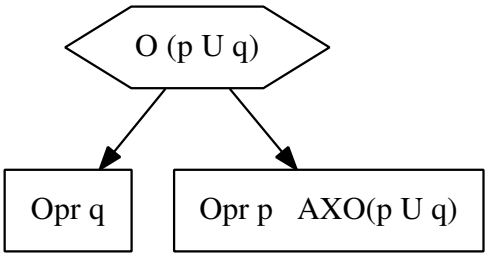

Figure 4.1: Expansion of an OR-node d with $L(d) = \{\mathbf{O}(\phi\, \mathcal{U}\, \psi)\}$.

Suppose that we are constructing the initial tableau T_0 and we have a frontier OR-node d. Then, we need to calculate the set of AND-node successors (denoted as $Blocks(d)$ in [Attie et al., 2004]) of d. This node d is *expanded* into a tree using the above characterization of nonelementary formulas as α or β. For instance, consider d labeled with $L(d) = \{\mathbf{O}(p\ \mathcal{U}\ q)\}$. This formula $f = \mathbf{O}(p\ \mathcal{U}\ q)$ is classified as a β formula, by definition $f \equiv \beta = Opr\ q\ \vee\ (Opr\ p\ \wedge\ \mathsf{AXO}(p\ \mathcal{U}\ q))$. So, for $\mathbf{O}(p\ \mathcal{U}\ q)$ two AND-node successors c and c' are generated, where $L(c) = \{Opr\ q\}$ and $L(c') = \{Opr\ p,\ \mathsf{AXO}(p\ \mathcal{U}\ q)\}$. This example is illustrated in Figure 4.1, where OR-nodes (d) are displayed as hexagons and AND-nodes (c and c') are displayed as rectangles. The successor c labeled with $Opr\ q$ certifies that the deontic eventuality $\mathbf{O}(p\ \mathcal{U}\ q)$ is fulfilled, while the successor c' labeled with $Opr\ p\ \wedge\ \mathsf{AXO}(p\ \mathcal{U}\ q)$ propagates $\mathbf{O}(p\ \mathcal{U}\ q)$. In general, given an OR-node d with $L(d) = \{f\}$, being $f \equiv \beta_1 \vee \beta_2$ a β formula, then add two AND-nodes successors c and c' to d with labels $L(c) = L(d) - \{f\} \cup \{\beta_1\}$ and $L(c') = L(d) - \{f\} \cup \{\beta_2\}$.

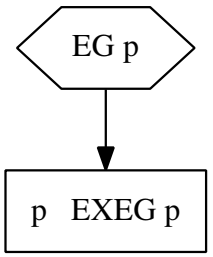

Figure 4.2: Expansion of an OR-node d with $L(d) = \{\mathsf{EG}\ p\}$.

On the other hand, consider d labeled with $L(d) = \{\mathsf{EG}\ g\}$. This formula $f = \mathsf{EG}\ g$ is classified as an α formula, by definition $f \equiv \alpha = g \wedge \mathsf{EXEG}\ g$. So, for $\mathsf{EG}\ g$ only one AND-node successor c is generated where $L(c) = \{g, \mathsf{EXEG}\ g\}$. This example

is illustrated in Figure 4.2. In general, given an OR-node d with $L(d) = \{f\}$, $f \equiv \alpha_1 \wedge \alpha_2$ being an α formula, then add a single AND-node successor c to d with label $L(c) = L(d) - \{f\} \cup \{\alpha_1, \alpha_2\}$.

4.2.3 Successors of AND-nodes

The set of OR-node successors of an AND-node c, denoted as $Tiles(c)$ in [Emerson and Clarke, 1982; Attie et al., 2004], are obtained as follow:

Let the set of formulas of c contain the following $\mathsf{AX}\ P_i$ and $\mathsf{EX}\ Q_j$ formulas:

$$\mathsf{AX}\ P_0, \ldots, \mathsf{AX}\ P_n \qquad \text{and} \qquad \mathsf{EX}\ Q_0, \ldots, \mathsf{EX}\ Q_m$$

Then, the successors of c are:

$$d_1 = \{P_0, \ldots, P_n, Q_0\}, \ldots, d_m = \{P_0, \ldots, P_n, Q_m\}$$

The above result assumes that c has at least one formula of the form $\mathsf{EX}\ Q_j$. Otherwise, we have to consider two particular cases. The first one, if c has no formulas of the form $\mathsf{AX}\ P_i$ and $\mathsf{EX}\ Q_j$, then a simple *dummy* of c is defined as successor of c, i.e., $Tiles(c) = \{d\}$ where $L(d) = \{f : f \in L(c)\}$. The second one, if the AND-node c has no formulas of the form $\mathsf{EX}\ Q_j$, then we add the formula $\mathsf{EX}\ True$ to the set of formulas of c ($L(c) = L(c) \cup \{\mathsf{EX}\ True\}$) and recompute $Tiles(c)$.

Let us now consider frontier AND-node c during the initial construction of the tableau T_0, where $L(c) = \{\mathsf{AXAF}\ q, p, \mathsf{E}(p\ \mathcal{U}\ q), \mathsf{EXE}(p\ \mathcal{U}\ q), \mathsf{EXEG}\ s\}$. According to the previous rule, c will have two OR-node successors d and d' labeled with $L(d) = \{\mathsf{AF}\ q, \mathsf{E}(p\ \mathcal{U}\ q)\}$ and $L(d') = \{\mathsf{AF}\ q, \mathsf{EG}\ s\}$ (see Figure 4.3).

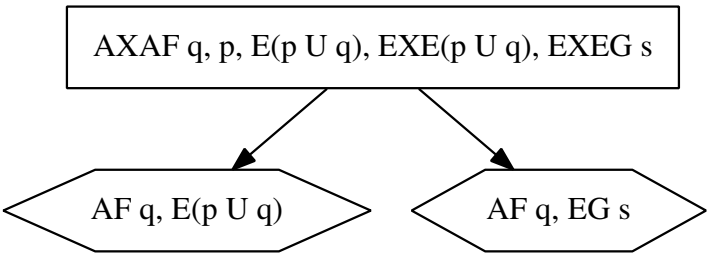

Figure 4.3: Tiles of an AND-node.

One difference of the above successor rules for AND-nodes and OR-nodes with respect to the ones in [Emerson and Clarke, 1982; Attie et al., 2004] is that in our case we do not consider process indexes indicating which nondeterministic choice is made in a concurrent program. In fact, we are interested only in the case of a single component.

Note that the tableau construction terminates when all leaf nodes are labeled only with elementary formulas. This is achieved because during the expansion of OR-nodes, one nonelementary formula f is deleted from an OR-node d and replaced by one or two smaller formulas in one or two successor AND-node respectively.

4.2.4 Pruning Rules

The next step of the dCTL- decision procedure is to remove certain inconsistent nodes of the tableau T_N. Firstly, we need to reproduce the next definition from [Attie et al., 2004]:

Definition 4.2.3. (Full subdag) A *full subdag* Q rooted at node e in a tableau T is a finite directed acyclic subgraph of T satisfying the following conditions:

(1) For each interior OR-node d in Q exactly one of its successor AND-nodes in T is in the subgraph.

(2) For each interior AND-node, all of its successor OR-nodes in T are in the subgraph.

(3) Node e is the unique node from which all other nodes in Q are reachable.

The following are the deletion rules applied to the tableau T_N:

- *RemoveP*: remove any node containing a formula and its negation.
- *RemoveOR-node*: remove an OR-node if all its original successors have been deleted.
- *RemoveAND-node*: remove an AND-node if one of its original successors has been deleted.

- *RemoveEF*: remove any node c such that $\mathsf{EF}(\phi) \in L(c)$ and it is not satisfiable, i.e., there does not exist a path from c to an AND-node c' such that $\phi \in L(c')$.

- *RemoveEU*: remove any node c such that $\mathsf{E}(\phi\ \mathcal{U}\ \psi) \in L(c)$ and it is not satisfiable, i.e., there does not exist a path from c to an AND-node c' such that $\psi \in L(c')$ and all other AND-nodes in the path contain the formula ϕ.

- *RemoveAF*: remove any node c such that $\mathsf{AF}(\phi) \in L(c)$ and there does exist a full subdag Q rooted at c such that for all nodes c' on the frontier of Q, $\phi \in L(c')$.

- *RemoveAU*: remove any node c such that $\mathsf{AF}(\phi\ \mathcal{U}\ \psi) \in L(c)$ and there does exist a full subdag Q rooted at c such that for all nodes c' on the frontier of Q, $\psi \in L(c')$ and for all interior AND-nodes c'' in Q, $\phi \in L(c'')$.

- *MarkFaultyPU*: mark any AND-node c as faulty such that $\mathbf{P}(\phi\ \mathcal{U}\ \psi) \in L(c)$ and it is not satisfiable, i.e., the tableau does not include a path from c to an AND-node c' such that $\psi \in L(c')$ and all other AND-nodes in the path contain the formula ϕ.

- *MarkFaultyOU*: mark any AND-node c as faulty such that $\mathbf{O}(\phi\ \mathcal{U}\ \psi) \in L(c)$ and there does exist a full subdag Q rooted at c such that for all nodes c' on the frontier of Q, $\psi \in L(c')$ and for all interior AND-nodes c'' in Q, $\phi \in L(c')$.

We remark that, the first seven rules are reproduced from [Emerson and Clarke, 1982; Attie et al., 2004] regarding CTL formulas, and we have defined the last two rules for considering the deontic eventuality formulas.

The process of pruning the tableau T_N finishes when none of the above rules is applicable. Roughly speaking, some of these rules remove all nodes that are either propositionally inconsistent, or do not have enough successors, or are labeled with an CTL eventuality formula which is not fulfilled. The presence of a suitable full subdag rooted at e serves to certify the fulfillment of the corresponding eventuality in $L(e)$. Note that the last two rules, *MarkFaultyPU* and *MarkFaultyOU*, do not remove any node, actually these mark the inconsistent nodes (those that do not fulfill a deontic eventuality) as faulty. The resulting faulty nodes will be treated during the process of checking the level of fault-tolerance (see Section 4.4). Finally, since each application

removes or marks one node, and T_N is finite, this process must terminate. Upon termination, if the root of T_N has been removed, then f_0 is unsatisfiable. Otherwise f_0 is satisfiable, in which case let T'_N be the tableau induced by the remaining nodes.

The last step of the decision procedure of dCTL- is to unravel the tableau into a model of f_0. We postpone the explanation of this step to Section 4.5, where we will include also the faulty part of the tableau (see the following Section 4.3).

4.3 Injection of Faults

In this section we describe our method for injecting *faults* into the initial tableau. This method is one of the steps during the synthesis process (see Section 4.4). We need to extend the Definition 4.2.1 in order to include the new kinds of nodes for considering the occurrences of faults.

Definition 4.3.1. (Faulty Graph). A faulty graph (tableau) T_F is a tuple with the following components:

(1) V_C, a set of AND-nodes,

(2) V_D, a set of OR-nodes,

(3) V_{FD}, a set of FOR-nodes,

(4) V_{FA}, a set of FAND-nodes,

(5) $A_{CD} \subseteq V_C \times V_D$, a set of AND-OR transitions,

(6) $A_{DC} \subseteq V_D \times V_C$, a set of OR-AND transitions,

(7) $A_{C-FD} \subseteq V_C \times V_{FD}$, a set of AND-FOR transitions,

(8) $A_{FD-FC} \subseteq V_{FD} \times V_{FC}$, a set of FOR-FAND transitions,

(9) $A_{FC-FC} \subseteq V_{FC} \times V_{FC}$, a set of FAND-FAND transitions,

(10) $A_{FC-C} \subseteq V_{FC} \times V_C$, a set of FAND-AND (recovery) transitions,

(11) $L : V_C \cup V_D \cup V_{FD} \cup V_{FC} \rightarrow 2^{cl(f)}$ is a labeling function which labels each node in $V_C \cup V_D \cup V_{FD} \cup V_{FC}$ with a subset of $cl(f)$.

Algorithm 5 Injection of faults via deontic propositional variables.

Require: AND/OR Tableau T_N
Ensure: Faulty Tableau T_F.
1: **for all** $andN \in V_C$ **do**
2: create FOR-node fo with $L(fo) = L(andN)$
3: attach fo as faulty successors of $andN$
4: **end for**
5: **for all** $fo \in V_{FD}$ (the set of FOR-nodes) **do**
6: **for all** $deonp_1 \in L(fo)$, being $deonp_1$ a deontic propositional variable **do**
7: create FAND-node fa with $L(fa) = L(fo)$
8: insert violation of $deonp_1$ on fa
9: attach fa as faulty successors of fo
10: create FAND-successors from fa with all possible combination of deontic prop. variables
11: **end for**
12: **end for**

A faulty tableau T_F is an AND/OR graph, but adding two new classes of nodes: FOR-nodes (Faulty OR-nodes) and FAND-nodes (Faulty AND-nodes) and four transition relations. We introduce some technical definitions which will be used in the following Section 4.4:

$$Succ_N(s) = \{s' \in V_C |\ \exists\ o \in V_D : (s, o) \in A_{CD} \wedge (o, s') \in A_{DC}\}$$

as the set of indirect non-faulty AND-node successors of s. Similarly, we denote by $Succ(s)$ the set of indirect AND-node successors of s, where these successors could be either faulty or not. Moreover, we represents the set of successors of s reachable via faulty arcs as follows:

$$\begin{aligned} Succ_F(s) = &\{s' \in V_{FA} |\ \exists\ o \in V_{FD} : (s, o) \in A_{C-FD} \wedge (o, s') \in A_{FD-FC}\} \cup \\ &\{s' \in V_{FA} |\ (s, s') \in A_{FC-FC}\} \end{aligned}$$

Analogously, we define $Pred_N(s')$ and $Pred_F(s')$ as the set of predecessors of s' via normal and faulty transitions, respectively.

The process of injecting faults starts when we have obtained a positive answer (the input specification is satisfiable) from the dCTL- decision procedure.

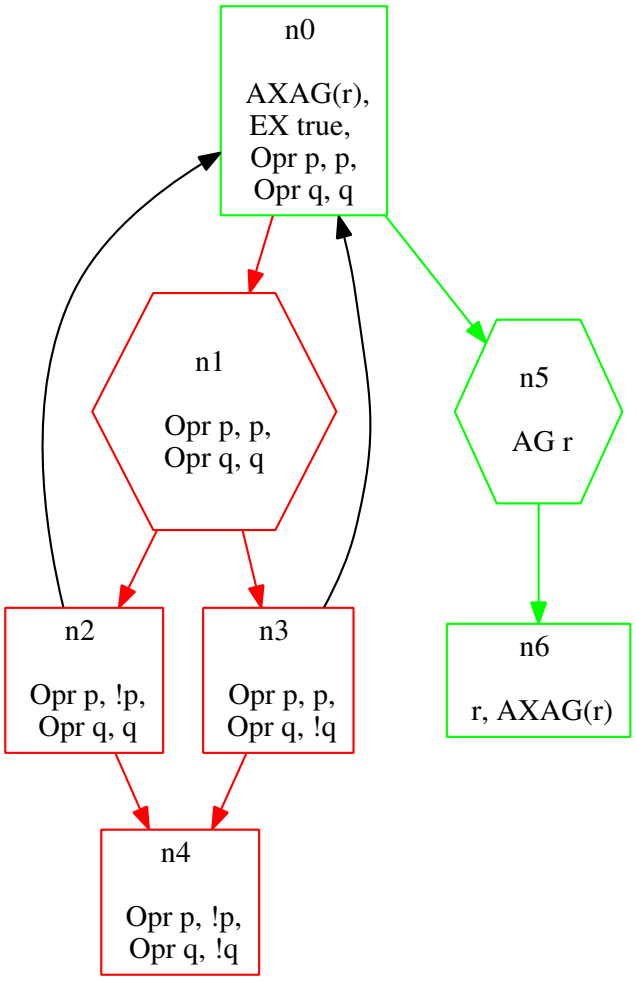

Figure 4.4: A part of a faulty tableau.

In order to have a better understanding of the injection mechanism, consider a part of a faulty tableau illustrated in Figure 4.4. In general, we take each non-faulty state (i.e., each AND-node drawn as green boxes) and produce a copy of it which is an FOR-node (faulty OR-nodes, shown as red hexagons), where we add a transition in A_{C-FD} from the AND-node to the new FOR-node. The corresponding FOR-node is labeled identically as its AND-node predecessor. From this FOR-node we generate all possible faults from deontic formula violations. Particularly, the AND-node n_0 has *Opr* p and *Opr* q, deontic propositional variables, expressing that p and q should be true there, which is the case in this node. Now, we start to consider those cases in which an obligation might be violated. Then, we negate one-by-one these deontic propositional variables and producing new FAND-nodes. As a result, two FAND-nodes n_2 and n_3 are generated from the FOR-node n_1 with similar information to it except for the new negated propositional variable. The FAND-nodes n_2 and n_3 introduce $\neg\ p$ and $\neg\ q$ violating *Opr* p and *Opr* q, respectively. We add transitions to A_{FD-FC} from n_1 to n_2 and n_3. Subsequently, from these two FAND-nodes a new FAND-node n_4 is create which combines both violations of *Opr* p and *Opr* q. We continue the process of negating deontic propositional variables from these FAND-nodes until there is no more possible combination of deontic propositional variables. This process must terminate due to the we have a finite number of deontic propositional variables for each AND-node in the tableau. This technique is sketched in Algorithm 5.

On the other hand, we allow users to provide their own fault specification, providing it as input to our synthesis algorithm. For example, $\mathsf{AG}(p \rightarrow \mathsf{AX}(\neg p))$ introduces a new FOR-node n_1 with the same information of the AND-node (n_0) which as a label p, then a faulty transition is attached from FOR-node n_1 to FAND-node n_2 in Figure 4.4. Similarly, a recovery transition from FAND-node n_3 to AND-node n_0 is introduced through $\mathsf{AG}(\neg q \rightarrow \mathsf{AX}(q))$. In general, these kinds of formulas have the form $\mathsf{AG}(cond \rightarrow \mathsf{AX}(prop\_formula))$, where *cond* and $prop\_formula$ are propositional formulas.

Finally, depending on the degree of fault-tolerance chosen by the user some of these FAND-nodes could be cut out from the tableau. Moreover, recovery transitions (shown as solid black lines in Figure 4.4) need to be added in order to go back to the normal behavior of the system and guarantee the desired level of fault-tolerance. We explain in more detail the corresponding algorithms for masking, nonmasking, and

failsafe in the next Section 4.4.

4.4 The Synthesis Method

In this section, we explain the details of our synthesis algorithm and how we deal with the three levels of fault-tolerance (masking, nonmasking, and failsafe). As we mention above, we require as input a deontic specification *dSpec*: *init-spec* and *normal-spec*, an optional fault specification *fSpec*: *fault-variables*, *fault-spec*, and *combine-spec*, a desired level of fault-tolerance, and the *interface*.

The pseudocode of our general synthesis algorithm is shown in Algorithm 6. It starts by building an initial tableau based on the dCTL- decision procedure defined in Section 4.2. That is, we construct a graph $T_N = (d, V_C, V_D, A_{CD}, V_{DC}, L)$. During the construction, AND-nodes and OR-nodes are expanded following the successor rules described in Subsection 4.2.3 and 4.2.2, respectively. Additionally, when a new AND-node (say c) is created, we check if there is some violation, i.e., if either $Opr\ p \in L(c)$ and $p \notin L(c)$, or $Opr\ \neg p \in L(c)$ and $p \in L(c)$, belong to the node. If this is the case, the node is considered faulty (proposition $Opr\ p$ is understood as: *p should be true*, and when p is false we get a state in which the normal or desirable behavior is not fulfilled). Otherwise, the node is added to $Norm$, the set of normal (non-faulty) states (line 10 of Algorithm 6). If there is a faulty node (say e') such that it has the same CTL formulas as a non-faulty node (say e''), e' is deleted, since it is masked by e'' (line 13). The process of construction stops when the set of *frontier* nodes is empty, i.e., all nodes have at least one successor. We then start applying the deletion rules (see Subsection 4.2.4), in order to remove inconsistent nodes and nodes containing eventuality formulas that cannot be satisfied. Moreover, during this process nodes originating from the specification of the system, that cannot fulfil deontic eventualities, are marked as faulty. We repeatedly apply the deletion rules until there is no change. When this process finishes, if the OR-node root d_0 has been removed, then *dSpec* is unsatisfiable, otherwise it is satisfiable. In the case that *dSpec* is satisfiable, let T'_N be the tableau induced by the remaining nodes. Assuming that the input specification *dSpec* is satisfiable, the next phase of our algorithm is the injection of faults into the tableau T'_N, obtaining a faulty tableau T_F. Part of this process was explained in Subsection 4.3 and it is sketched in Algorithm 5. Additionally, we need

Algorithm 6 Construction of Faulty Tableau T_F.

Require: *dSpec* : (*init-spec*, *normal-spec*), $fSpec$: (*fault-variables*, *fault-spec*, *combine-spec*), level-of-tolerance, *interface* $Intf$

Ensure: Faulty tableau T_F satisfying the level of fault-tolerance

1: Let d be an OR-node with label $\{dSpec\}$
2: $T_N := d$
3: **repeat**
4: Select a node $e \in frontier(T_N)$
5: **if** $\exists\ e' \in V_D$ with $L(e) = L(e')$ **then**
6: merge e and e'
7: **else**
8: **for all** $e' \in Succ(e)$ being an AND-node **do**
9: **if** e' is non-faulty **then**
10: $Norm := Norm \cup \{e'\}$
11: **else**
12: **if** $\exists e'' \in Succ(e)$ faulty such that $NForm(e') = NForm(e'')$ **then**
13: delete(e")
14: **end if**
15: **end if**
16: **end for**
17: attach all $e' \in Succ(e)$ as successors of e and mark e as expanded
18: **end if**
19: update V_C, V_D, A_{CD}, V_{DC} appropriately
20: **until** $frontier(T_N) = \emptyset$
21: **repeat**
22: Apply deletion rules
23: **until** T_N does not change
24: Apply Algorithm 5 and 7 for injecting faults
25: Apply corresponding simulation algorithm for the given level-of-tolerance
26: **return** T'_F

to consider those faults provided by the user in the input fault specification $fSpec$. These are introduced following the Algorithm 7.

Algorithm 7 Injection of faults from the fault specification $fSpec$.

Require: Tableau T_N generated by Algorithm 6, $fSpec$: (*fault-variables*, *fault-spec*, *combine-spec*)
Ensure: Faulty Tableau T_F.

```
1: for all f ∈ fault-spec, where f has the form AG(cond → AX(prop_formula)) do
2:   for all andN ∈ Norm do
3:     if cond(f) ∈ L(andN) then
4:       create FOR-node fo with L(fo) = L(andN)
5:       attach fo as faulty successors of andN
6:       create FAND-node fa with L(fa) = (L(fo)\{cond(f)})∪{prop_formula}
7:       attach fa as faulty successors of fo
8:     end if
9:   end for
10: end for
11: for all f ∈ fault-spec do
12:   for all fandN ∈ V_FA do
13:     if cond(f) ∈ L(fandN) then
14:       add recovery transition from fandN to each Norm state n, with
          prop_formula ∈ L(n)
15:     end if
16:   end for
17: end for
```

Subsequently, we apply the corresponding masking, nonmasking, or failsafe simulation algorithms (see below), depending on the user-defined level of fault-tolerance and the defined interface. During this phase, considering the case that the user wants to obtain a masking fault-tolerant program, only those nodes that can be masked are preserved, and we cut out the remaining ones (obtaining T'_F). Finally, in the last phase of our synthesis method, we extract from the generated faulty tableau T'_F (see Section 4.5) the synthesized fault-tolerant program satisfying the required level of fault-tolerance.

4.4.1 Synthesis Algorithm for Masking Fault-Tolerance

In this subsection, we present the algorithm for masking fault-tolerance. In general, for the three levels of fault-tolerance, the main idea is to use the efficient procedures presented in Chapter 3, Section 3.5 for computing masking, nonmasking, and failsafe relations. In those algorithms, a colored Kripke structure is required as input, which is the combination of a system specification M and a fault-tolerant implementation M' in a single structure $M \oplus M'$ via disjoint union. In this process, we need to apply those algorithms to a faulty tableau T_F and the input *interface*.

For this case, the process for computing a masking relation over a faulty tableau T_F is sketched in Algorithm 8. A difference between Algorithm 8 and 2 is the predecessor and successor operations used, e.g. $Succ_N(s)$ are used in Algorithm 8 instead of $Post_N(s)$ in Algorithm 2.

The algorithm generates a masking relation $Mask$ satisfying conditions $B.3$ and $B.4$ of Definition 3.1.1. Note that the loop of line 27 finally looks for those states which are not in the relation $Mask$. Consequently, these states are deleted from the current faulty tableau together with the directed acyclic graphs which have as root each of these states. Additionally, condition $B.2$ of Definition 3.1.1 is computed using Algorithm 9. This step may also lead to cutting out further faulty nodes, namely those which exhibit normal behavior, but that are not part of the correct behavior of the system. Hence, we obtain a faulty tableau T'_F satisfying the conditions of Definition 3.1.1.

Let us state two important properties of the synthesis algorithm, whose proofs are sketched following the proofs of correctness given for the algorithms for CTL satisfiability based on tableau [Clarke and Emerson, 1981; Attie et al., 2004], and for checking (bi)simulations [Baier and Katoen, 2008; Henzinger et al., 1995].

Algorithm 8 Algorithm Masking: compute conditions $B.3$ and $B.4$ of Definition 3.1.1

Require: Faulty Tableau T_F generated by Algorithm 6 and an *interface* $Intf$.
Ensure: Conditions $B.3$ and $B.4$ hold.

1: **for all** $s_2 \in V_C \cup V_{FA}$ **do**
2: $Mask(s_2) := \{s_1 \in Norm \mid Intf(s_1) = Intf(s_2)\}$
3: $RemoveL(s_2) := Norm \backslash Pred_N(Mask(s_2))$ {Note that all the nodes in $Norm$ are already generated}
4: **end for**
5: **while** $\exists\ s_2' \in S \backslash Norm$ with $RemoveL(s_2') \neq \emptyset$ **do**
6: select s_2' such that $RemoveL(s_2') \neq \emptyset$
7: **for all** $s_1 \in RemoveL(s_2')$ **do**
8: **for all** $s_2 \in Pred_N(s_2')$ **do**
9: **if** $s_1 \in Mask(s_2)$ **then**
10: $Mask(s_2) := Mask(s_2) \backslash \{s_1\}$
11: **for all** $s \in Pred_N(s_1)$ with $Succ_N(s) \cap Mask(s_2) = \emptyset \wedge s \notin Mask(s_2)$ **do**
12: $RemoveL(s_2) := RemoveL(s_2) \cup \{s\}$
13: **end for**
14: **end if**
15: **end for**(* this takes care of the faulty transitions*)
16: **for all** $s_2 \in Pred_F(s_2')$ **do**
17: **if** $s_1 \in Mask(s_2) \wedge s_1 \notin Mask(s_2')$ **then**
18: $Mask(s_2) := Mask(s_2) \backslash \{s_1\}$
19: **for all** $s \in Pred_N(s_1)$ with $Succ_N(s) \cap Mask(s_2) = \emptyset$ **do**
20: $RemoveL(s_2) := RemoveL(s_2) \cup \{s\}$
21: **end for**
22: **end if**
23: **end for**
24: **end for**
25: $RemoveL(s_2') := \emptyset$
26: **end while**
27: **for all** s_2 **do**
28: **if** $Mask(s_2) = \emptyset$ **then**
29: delete $DAG[s_2]$
30: **end if**
31: **end for**

Algorithm 9 Computes relations that satisfy condition $B.2$ of Definition 3.1.1

Require: Faulty Tableau T_F generated by Algorithm 6 and an *interface*
Ensure: Relations $Masked$ and $RemoveR$ satisfy condition $B.2$ of Definition 3.1.1

1: **for all** $s_2 \in V_C$ **do**
2: $Masked(s_2) := \{s_1 \mid Intf(s_1) = Intf(s_2)\}$
3: $RemoveR(s_2) := S \backslash Pre(Masked(s_2))$ {Note that all the faulty and normal states are already generated}
4: **end for**
5: **while** $\exists\ s_2' \in V_C$ with $RemoveR(s_2') \neq \emptyset$ **do**
6: select s_2' such that $RemoveR(s_2') \neq \emptyset$
7: **for all** $s_1 \in RemoveR(s_2')$ **do**
8: **for all** $s_2 \in Pre(s_2')$ **do**
9: **if** $s_1 \in Masked(s_2)$ **then**
10: $Masked(s_2) := Masked(s_2) \backslash \{s_1\}$
11: **for all** $s \in Pre(s_1)$ with $Post(s) \cap Masked(s_2) = \emptyset$ **do**
12: $RemoveR(s_2) := RemoveR(s_2) \cup \{s\}$
13: **end for**
14: **end if**
15: **end for**
16: **end for**
17: $RemoveR(s_2') := \emptyset$
18: **end while**
19: **for all** $s_2 \in V_C \cup V_{FA}$ **do**
20: **if** $Masked(s_2) = \emptyset$ **then**
21: delete $DAG[s_2]$
22: **end if**
23: **end for**

Theorem 4.4.1. *Given a specification S over a set AP of propositional letters, if we obtain a program P by applying the synthesis algorithm over the sublabeling obtained from $AP' \subseteq AP$, then P is a masking tolerant implementation of S, i.e., $P \prec_{Mask} \ P$ (with respect to AP') and $P \vDash S$.*

Proof. First, we prove that Algorithm 8 ensures conditions $B.3$ and $B.4$ of Definition 3.1.1. Then we prove that Algorithm 9 ensures condition $B.2$. Notice that, when Algorithm 8 starts, all the normative nodes ($Norm$) have been computed. Then, we have the following invariant for Algorithm 8: *(i)* $RemoveL(s_2) = Norm \backslash Pre_N(Mask(s_2))$; *(ii)* for any relation $\prec_{Mask}$: $\{s_1 \in Norm \mid s_1 \prec_{Mask} s_2\} \subseteq Mask(s_2) \subseteq \{s_2 \in Norm \mid Intf(s_1) = Intf(s_2)\}$; and *(iii)* $\forall s_2 \in Mask(s_1)$, either:

- $\exists s'_1 \in Succ(s_1)$ with $Succ_N(s_2) \cap Mask(s'_1) = \emptyset \wedge s'_1 \notin Mask(s_1)$ and $s_2 \in RemoveL(s_1)$,
- $\forall s'_1 \in Succ(s_1) : Succ_N(s_2) \cap Mask(s'_1) = \emptyset$

From the last item we obtain that, when $RemoveL(s'_1) = \emptyset$ for every s'_1, then: $\forall s_1 \in S : \forall s_2 \in Mask(s_1) : \forall s'_1 \in Post(s_1) : Succ_N(s_2) \cap Mask(s'_1) \neq \emptyset \vee s_2 \in Mask(s'_1)$. That is, for the relation defined as $s_1 \prec_{Mask} s_2$, items $B.3$ and $B.4$ of Definition 3.1.1 hold.

On the other hand, notice that before executing Algorithm 8, all the faulty nodes have been calculated. For this algorithm, we have the following invariant:
(i) $RemoveR(s_2) = S \backslash Pred(Masked(s_2))$; *(ii)* for any relation $\prec_{Mask}$: $\{s_2 \in V_C \mid s_1 \prec_{Mask} s_2\} \subseteq Masked(s_1) \subseteq \{s_2 \in V_c \mid Intf(s_1) = Intf(s_2)\}$; and *(iii)* $\forall s_2 \in Masked(s_1)$, either:

- $\exists s'_1 \in Succ(s_1)$ with $Succ(s_2) \cap Masked(s'_1) = \emptyset$ and $s_2 \in RemoveL(s_2)$,
- $\forall s'_1 \in Succ_N(s_1) : Succ(s_2) \cap Masked(s'_1) = \emptyset$

That is, when $RemoveR(s_1) = \emptyset$, then we have $\forall s_1 \in S : \forall s_2 \in Masked(s_1) : \forall s'_1 \in Succ_N(s_1) : Succ(s_2) \cap Masked(s'_1) \neq \emptyset$. Thus, the relation defined as: $s_1 \prec_{Mask} s_2$ iff $s_1 \in Mask(s_2) \wedge s_2 \in Masked(s_1)$, satisfies condition $B.2$ of Definition 3.1.1. Since it also satisfies $B.3$ and $B.4$, it is a masking relation. The proof that the obtained structure satisfies the specification can be obtained, for CTL operators, following the

proof given in [Attie et al., 2004]. For the deontic operators notice that all the nodes that do not satisfy the deontic operators are marked as faulty, ensuring that the safety deontic formulas are preserved. We treat deontic eventualities by marking as faulty all the nodes that have unfulfilled deontic eventualities. Thus, both CTL and deontic formulas are satisfied. Termination can be proved by resorting to the approach for proving termination of simulation algorithms (cf. [Baier and Katoen, 2008]). The only point to note is that the injection of faults finishes at some point since states start repeating.

□

The definition of masking similarity ensures that the safety and liveness properties of the normal behavior of P are preserved in the presence of faults. If the synthesized program P contains no faults, we conclude that it is not possible to synthesize a masking tolerant program supporting faults, from the specification. Moreover, we can prove that the synthesized program is the most general satisfying these properties.

Theorem 4.4.2. *Given a specification S, if a structure M is obtained by the synthesis algorithm, then for any other structure $M' \vDash S$ such that it is masking and the non-faulty part of M' coincides with that of M, then we have $M' \prec M$, where $\prec$ is the usual notion of simulation with respect to the* interface $Intf$.

Proof. The simulation relation is defined as: *(i)* if $s \in Norm$, $s' \prec s$ iff $s' \prec_{Mask} s$; *(ii)* if $s \notin Norm$, $s' \prec s$ iff $Masked(s') \subseteq Masked(s)$, i.e., the M' nodes masked for s' are a subset of those masked by s in M. In order to prove that this relation is a simulation, assume $s \prec t$. If $s \rightarrow s'$ and $s' \in Succ_N(s)$, by condition $B.3$ of Definition 3.1.1 we obtain that there is a $t \rightarrow t'$ such that $s' \prec t'$. Otherwise, if $s \rightarrow s'$ and $s' \in Succ_F(S)$ and s is normative, then the transition matches some part of the specification. Thus, a similar transition is in M and therefore we have $t \rightarrow t'$, now if s' masks any node, the same node have to be masked by t' (otherwise M' would not be masking) thus $s' \prec t'$. A similar reasoning can be used when s is faulty.

□

As a corollary of this Theorem, the synthesis algorithm is complete. The CTL algorithm is complete, i.e., if some structure that satisfies the CTL specification exists, then the algorithm produces it, and by Theorem 4.4.2 we obtain a program that is masking, and preserves as many faulty states as possible.

4.4.2 Synthesis Algorithm for Nonmasking Fault-Tolerance

We now present the algorithm for computing nonmasking fault-tolerance. This is sketched in Algorithm 10, where it is an adjustment to Algorithm 3 considering a faulty tableau T_F and the input *interface* $Intf$.

Roughly speaking, this algorithm explores the faulty tableau T_F and checks whether conditions $B.3$, $B.4$ and $B.5$ of Definition 3.2.1 are satisfied, considering the input interface $Intf$. As a result, a relation $NMask$ is returned, which consists of those pairs of states that satisfy these conditions. Subsequently, we have to update the faulty tableau and prune those states that are not in the relation. More specifically, for each $s_2 \in V_C \cup V_{FA}$, if $NMask(s_2) = \emptyset$, then we remove the direct acyclic graph rooted at s_2. Additionally, we have to check condition $B.2$ of Definition 3.2.1. Notice that it is similar to condition $B.2$ of Definition 3.1.1, and hence we can test this condition using Algorithm 9. Hence, we obtain a fault tableau satisfying all the conditions of Definition 3.2.1.

We have presented the correctness of computing a nonmasking relation from a colored Kripke structure in Subsection 3.5.2. For the case of a faulty tableau T_F, the correctness proof is similar to the the proof in Subsection 3.5.3. Analogously, for the proof of the complexity of the algorithm.

4.4.3 Synthesis Algorithm for Failsafe Fault-Tolerance

We finally present an algorithm (see Algorithm 11) to calculate a relation of failsafe fault-tolerance. This is adapted version of Algorithm 4 for computing a faulty tableau T_F taking into account the input *interface*. The scheme of this algorithm is similar to that of Algorithm 8. This is because both masking and failsafe fault-tolerance require that the safety properties have to be guaranteed. However, liveness properties are not necessarily preserved in failsafe fault-tolerance.

In Subsection 3.5.3, we have proved the correctness of computing a failsafe relation from a colored Kripke structure; the proof for the case of a faulty tableau T_F is straightforward, following the proof in Subsection 3.5.3. Similarly, for the proof of the complexity of the algorithm.

Algorithm 10 Computation of nonmasking fault-tolerant

Require: Faulty tableau T_F generated by Algorithm 6 and an *interface*
Ensure: Conditions $B.3$, $B.4$ and $B.5$ of Definition 3.2.1 are checked.

for all $s_2 \in V_C \cup V_{FA}$ **do**
  $NMask(s_2) := \{s_1 \in Norm \mid Intf(s_1) = Intf(s_2)\}$
  $Remove(s_2) := Norm \backslash Pred_N(NMask(s_2))$
  $Remove^+(s_2) := Norm \backslash Pred_N(NMask(Post^*(s_2)))$
end for
while $\exists\ s'_2 \in V_C \cup V_{FA}$ with $Remove(s'_2) \cup Remove^+(s'_2) \neq \emptyset$ **do**
  select s'_2 such that $Remove(s'_2) \cup Remove^+(s'_2) \neq \emptyset$
  for all $s_1 \in Remove(s'_2) \cup Remove^+(s'_2)$ **do**
    if $s_1 \in Remove(s'_2)$ **then**
      for all $s_2 \in Pred_N(s'_2)$ **do**
        if $s_1 \in NMask(s'_2)$ **then**
          $NMask(s_2) := NMask(s_2) \backslash \{s_1\}$
          for all $s \in Pred_N(s_1)$ with $Post_N(s) \cap NMask(s_2) = \emptyset \wedge (s \notin NMask(Pre_F(s_2)) \vee s_1 \in RemoveF(s'_2))$ **do**
            $Remove(s_2) := Remove(s_2) \cup \{s\}$
            if $s_1 \in NMask(s_2) \wedge s_1 \in Remove^+(s'_2)$ **then**
              $Remove^+(s_2) := Remove^+(s_2) \cup \{s\}$
            end if
          end for
        end if
      end for
    end if
    if $s_1 \in Remove^+(s'_2)$ **then**
      for all $s_2 \in Pred_F(s'_2)$ **do**
        if $s_1 \in NMask(s'_2)$ **then**
          $NMask(s_2) := NMask(s_2) \backslash \{s_1\}$
          for all $s \in Pred_N(s_1)$ with $Succ_N(s) \cap NMask(Post^*_F(s_2)) = \emptyset$ **do**
            $Remove^+(s_2) := Remove(s_2) \cup \{s\}$
            $Remove(s_2) := Remove(s_2) \cup \{s\}$
          end for
        end if
      end for
    end if
  end for
  $Remove(s_2) := \emptyset$
end while
return $\{\langle s_1, s_2 \rangle \mid s_1 \in NMask(s_2)\}$

Algorithm 11 Computation of failsafe fault-tolerant

Require: Faulty tableau T_F generated by Algorithm 6 and an *interface*
Ensure: failsafe fault-tolerant $\prec_{Failsafe}$

1: **for all** $s_2 \in V_C \cup V_{FA}$ **do**
2: $FSafe(s_2) := \{s_1 \in Norm \mid Intf(s_1) = Intf(s_2)\}$
3: $Remove(s_2) := Norm \backslash Pred_N(FSafe(s_2))$
4: **end for**
5: **while** $\exists\ s'_2 \in V_C \cup V_{FA}$ with $Remove(s'_2) \neq \emptyset$ **do**
6: select s'_2 such that $Remove(s'_2) \neq \emptyset$
7: **for all** $s_1 \in Remove(s'_2)$ **do**
8: **for all** $s_2 \in Pred_N(s'_2)$ **do**
9: **if** $s_1 \in FSafe(s_2)$ **then**
10: $FSafe(s_2) := FSafe(s_2) \backslash \{s_1\}$
11: **for all** $s \in Pred_N(s_1)$ with $Succ_N(s) \cap FSafe(s_2) = \emptyset \wedge (s \notin FSafe(Pre_F(s_2)) \vee Intf(s) \neq Intf(s'_2))$ **do**
12: $Remove(s_2) := Remove(s_2) \cup \{s\}$
13: **end for**
14: **end if**
15: **end for**
16: **for all** $s_2 \in Pred_F(s'_2)$ with $Intf(s_2) \neq Intf(s'_2)$ **do**
17: **if** $s_1 \in FSafe(s_2)$ **then**
18: $FSafe(s_2) := FSafe(s_2) \backslash \{s_1\}$
19: **for all** $s \in Pred_N(s_1)$ with $Succ_N(s) \cap FSafe(s_2) = \emptyset$ **do**
20: $Remove(s_2) := Remove(s_2) \cup \{s\}$
21: **end for**
22: **end if**
23: **end for**
24: **end for**
25: $Remove(s_2) := \emptyset$
26: **end while**
27: **for all** $s_2 \in V_C \cup V_{FA}$ **do**
28: **if** $FSafe(s_2) = \emptyset$ **then**
29: delete $DAG[s_2]$
30: **end if**
31: **end for**

4.5 Extraction of the Model from the Tableau

In this section we explain the last step of our synthesis approach, that is, the extraction of a model from a faulty tableau.

Let T'_F be the faulty tableau of T_F that remains after all nodes have been deleted applying the deletion rules and the corresponding simulation algorithm depending on the user-defined level of fault-tolerance. We will extract a finite model M_F of f_0 by a process of "unraveling", following the idea from [Attie et al., 2004; Emerson and Clarke, 1982] and adapting it to our approach.

Initially, we focus on the normal part of the tableau, i.e., those non-faulty AND-nodes in T'_F. Then, for each non-faulty AND-node c in T'_F, and for each eventuality formula $g \in L(c)$, there is a subdag, $DAG[c, g]$, rooted at c which guarantees that g is fulfilled. The idea is to employ these subdags to produce, for each AND-node c, a model of normal fragment $NFRAG[c]$ such that every eventuality in c is fulfilled within $NFRAG[c]$. Additionally, we have to include the faulty part of the tableau, those FAND-nodes in T'_F that satisfy the level of tolerance chosen by the user. Intuitively, we have to attach to each $NFRAG[c]$ those faults that can deviate from the normal behavior of the system starting from c. Finally, we consolidate these fragments to obtain a fault model M_F.

In the following two subsections, we explain this process in more detail.

4.5.1 Construction of fragments

For each AND-node c in T'_F, a normal fragment $NFRAG[c]$ is constructed. $NFRAG[c]$ is a directed acyclic graph whose nodes are non-faulty AND-nodes, and whose local structure is taken from the normal part of T'_F, that is, $c \rightarrow c'$ in $NFRAG[c]$ only if $c \rightarrow d \rightarrow c'$ in T'_F for some OR-node d. In addition, c is the root of $NFRAG[c]$ and all eventualities in the label of c are fulfilled in $NFRAG[c]$. Given that c was not removed by the deletion rules, it follows that, for each eventuality $g \in L(c)$, T'_F contains a full subdag with root c and in which g is fulfilled. Let $DAG[c, g]$ be the directed acyclic prestructure that results from removing all the OR-nodes from this subdag, and connecting the AND-nodes up appropriately, that is, if $c \rightarrow d$ and $d \rightarrow c'$ are edges in the full subdag for some OR-node d, then $c \rightarrow c'$ is an edge in $DAG[c, g]$. We construct $NFRAG[c]$ as follows. Let $g_1, \ldots, g_m$ be all of the eventualities in $L(c)$. Let

$NFRAG_1$ be a copy of $DAG[c, g_1]$. In order to obtain $NFRAG_{i+1}$ from $NFRAG_i$, do the following:

for all $c' \in frontier(NFRAG_i)$ **do**
  if $g_{i+1} \in L(c')$ **then**
    attach a copy of $DAG[c, g_{i+1}]$ to $NFRAG_i$ at c'
  end if
end for

Finally, let $NFRAG[c] = NFRAG_m$.

In the case that $L(c)$ does not contain eventualities, then $NFRAG'[c]$ consists of c together with enough local successors to satisfy all of the formulas that have form AX or EX as main connective. That is, for each $d \in Tiles(c)$, choose a $c' \in Blocks(d)$ and add c' as a successor of c. Identify all such c' with the same label. Then $NFRAG'[c]$ consists of c together with all such c'.

Subsequently, we have to attach the faults that affect each non-faulty AND-node c in T'_F. We then construct faulty fragments $FFRAG[c]$ from each $NFRAG[c]$ as follows.

for all $c' \in NFRAG[c]$ **do**
  for all FOR-node d such that $c' \dashrightarrow d$ **then**
    attach a copy of each $DAG[fa]$ to c', where $d \dashrightarrow fa$ in T'_F
  end for
end for

Note that each fa is a FAND-node, which is a direct successor from FOR-node d in T'_F. Intuitively, this algorithm attaches to each c' in $NFRAG[c]$ the direct acyclic graphs rooted at each fa, which contain only those FAND-nodes that have remained after the application of the corresponding simulation algorithm.

4.5.2 Construction of the model

The last step of this process is to connect all $FFRAGs$ calculated above. Basically, the frontier nodes of a particular $FFRAG$ are identified with root nodes of other $FFRAGs$ to form a particular Kripke structure (see below). The resulting structure is a model of *dSpec*. The procedure is as follows:

Choose $c_0 \in Blocks(d_0)$ arbitrarily, where d_0 is the root of T_F. Let $M_1 = FFRAG[c_0]$. In order to obtain M_{i+1} from M_i, do the following:

for all $s \in frontier(M_i)$ **do**
  if there exists $s' \in interior(M_i)$, such that s is also a copy of c, and
    a copy of $FFRAG[c]$ is directly embedded in M_i with root s **then**
      identify s and s'
  else
    replace s by a copy of $FFRAG[c]$
  end if
end for

The construction halts with $i = n$ when $frontier(M_n)$ is empty. Let $M_F = M_n$. We write $M_F = (c_0, \mathcal{N}, \mathcal{F}, R_N, R_F, R_{Rec}, L)$, where $c_0 \in Blocks(d_0)$ is chosen arbitrarily in M_F, $\mathcal{N}$ is a set of *normal*, or "green" states, $\mathcal{F}$ is a set of *faulty*, or "red" states, $R_N \subseteq \mathcal{N} \times \mathcal{N}$ is a (normal) transition relation, $R_F \subseteq \mathcal{N} \times \mathcal{F} \cup \mathcal{F} \times \mathcal{F}$ is a (faulty) transition relation, $R_{Rec} \subseteq \mathcal{F} \times \mathcal{N}$ is a (recovery) transition relation, and L is a labeling function indicating which propositions are true in each state. Finally, we restrict L to L_0 containing only the propositions occurring in *dSpec* and *fSpec*, i.e., $M_F = (c0, \mathcal{N}, \mathcal{F}, R_N, R_F, R_{Rec}, L_0)$ where $L_0 \subseteq L$. M_F is a model of *dSpec*.

4.6 Complexity of the Synthesis Method

In this section we give the time complexity of our synthesis method in terms of two parameters, following the proof given in [Attie et al., 2004]. Firstly, the length of the deontic specification *dSpec*: *init-spec* and *normal-spec* and secondly, the length of the fault specification *fSpec*: *fault-variables*, *fault-spec*, and *combine-spec*. We assume that each auxiliary atomic proposition in $fault-variables$ is specified at least once in

combine-spec (since otherwise it can be removed from the fault specification without changing the specification logically), and so the size of the set of auxiliary atomic propositions is always smaller than the size of the *combine-spec* specification. Hence it is not included as a parameter in the complexity analysis.

The construction of the initial tableau T_N involves creating nodes whose labels are subsets of $cl(dSpec)$. Thus, the number of nodes in the construction of the normal part of the tableau T_N is $\mathcal{O}(exp(|cl(dSpec)|))$, where $exp(n) \stackrel{\mathsf{def}}{=} 2^n$. Moreover, for any node e of T_N, $L(e) \subseteq cl(dSpec)$. Therefore, $|L(e)| \leq |cl(dSpec)|$. Hence, the sum of the lengths of the formulas in $L(e)$ is in $\mathcal{O}(|dSpec|^2)$, since each such formula has length in $O(|dSpec|)$. Thus, the size of each node in T_N is $\mathcal{O}(|dSpec|^2)$.

Notice that during the construction of the normal part of the tableau T_N, each node (either AND-node or OR-node) e in T_0 is expanded once. The process of expanding nodes essentially concern the computation of $Blocks(e)$ or $Tiles(e)$. On the one hand, generating $Tiles(e)$ has cost at most $|L(e)|$, because each formula in $L(e)$ adds at most one element of $Tiles(e)$. Hence, the cost is $\mathcal{O}(|dSpec|)$. On the other hand, the generation of $Blocks(e)$ requires the construction of a tree, where $\alpha - \beta$ rules are applied until all the leaves are labeled only with elementary formulas. The cost of this is at most $\mathcal{O}(|dSpec|^2 \sum_{f \in L(e)} |f|)$, since each $\alpha - \beta$ expansion delivers one connective in one formula in $L(e)$, and produces either one or at most two new nodes, each of size $\mathcal{O}(|dSpec|^2)$. Therefore, the total cost of calculating $Blocks(e)$ is $\mathcal{O}(|dSpec|^4)$, because each formula in $L(e)$ has length $\mathcal{O}(|dSpec|)$, and there are $\mathcal{O}(|dSpec|)$ such formulas.

Additionally, we need to consider the faults that can deviate from the normal behavior of the system. For each AND-node e in T_N, faults are injected into e via the violation of deontic propositional variables and from the fault specification $fSpec$. In the former, for each AND-node c, a new FOR-node fn is created and attached to c. Subsequently, FAND-nodes are generated from fn with all combinations of violating deontic propositional variables in fn. The number of deontic propositional variables in an AND-node c is significantly small compared with labels that appear in a node. This process can be performed in $\mathcal{O}(exp(|deonV(c)|))$ steps, where $deonV(c)$ is the number of deontic propositional variables in c. In the latter, for each AND-node c, we have to check if there exists any formula f in $fault - spec$, in which its antecedent is true in $L(c)$. If this is the case, a new FAND-node fc is attached to c labeled with the

same propositions of c, except that the antecedent of c is replaced by the consequent of f. This can be done in $|fault - spec|$ steps.

After the injection of faults, we have to prune those faulty states that do not satisfy the required level of fault-tolerance considering the input *interface* $Intf$. We presented for each level of fault-tolerance a simulation algorithm, which we apply for this purpose. The complexity times for all of these algorithms are polynomial with respect to the number of edges and the number of nodes of the the tableau T_F. More specifically, for masking fault-tolerance the simulation algorithm is computed in a running time of $\mathcal{O}(|S|^2 * |Intf| + |E| * |S|)$, where $|E|$ is the number of edges of the tableau and $|S|$ the number of states (all types of nodes in T_F). In the case of nonmasking fault-tolerance, its complexity time results is $\mathcal{O}(|S|^4 + |S|^2 * |AP'|)$. Finally, the time complexity for failsafe fault-tolerance is $\mathcal{O}(|E| * |S| + |E| * |Intf|)$

We need to consider other complementaries steps during our synthesis process, e.g., application of pruning rules, construction of normal and faulty fragments, and a posteriori construction of the final fault-tolerant model from these fragments. The complexity times of these steps are all polynomial in the size of T_F, i.e., $exp(\mathcal{O}(|dSpec|))$. The proofs of these steps are identical as that for the CTL decision procedure, which can be consulted in [Emerson, 1981].

In conclusion, the overall time complexity is $|fault - spec| * exp(\mathcal{O}(|dSpec|))$, that is, a single exponential in the specification size and linear in the size of the fault specification. We can also conclude from the above discussion that the overall space complexity is $|fault - spec| * exp(\mathcal{O}(|dSpec|))$.

In general, as stated in [Attie et al., 2004], *"synthesis methods based on exhaustive state exploration will have a time complexity no better than single exponential in the size of the specification"*.

Chapter 5

Case Studies

In this chapter we provide several case studies to validate our synthesis method. We first focus on examples of masking fault-tolerance. Typically, masking fault-tolerant programs involve redundancy, consensus, agreement, and majority voting techniques. We have performed different experiments based on some of these techniques such as, Byzantine agreement, N-Modular-Redundancy (NMR), a Memory Cell, and a Token ring. Additionally, we synthesized a simple Train System requiring failsafe as the level of fault-tolerance. Regarding nonmasking fault-tolerance, we provide two case studies, an Altitude Switch (ASW) controller and the Muller C-element.

We remark that, for the following cases studies, we do not present the complete resulting faulty tableau and the final models. This is because many of them are not visualized properly in this thesis's format. For that reason we provide portions of the tableau and the models of these case studies. At the end of this chapter, on Section 5.8.1, we provide information on how to access the full results for the following examples.

Finally, we introduce the architecture of our tool syntdctl. Moreover, we present the experimental results for some of the models described below.

5.1 A Memory Cell

We start with a simple memory cell example introduced in the Example 3.1.1. We rehearse this case study for the reader, including some extra details for the specification in the logic dCTL-.

The memory cell stores a bit of information, and supports reading and writing; redundancy, via three memory bits instead of one, is employed in order to deal with the situation in which a bit's value unexpectedly loses its charge and it turns into another value.

In order to specify this behavior in dCTL-, we use a variable v, that indicates the value that the user wants to write (i.e., $v = 0$, $v = 1$ or $v = \perp$, the latter being the case in which the system is "idle" with respect to writing) is added to the model. Writing operations are performed simultaneously on the three bits, whereas a reading returns the value that is repeated at least twice in the memory bits. For this problem, each state in the model is described by variables r_i and w_i (for $i = 0, 1$) which record the last writing operation performed and the actual reading in the state. Each state also has three bits, described by boolean variables c_0, c_1 and c_2. The requirements on this system (*dSpec* and *interface*) can be specified in dCTL- as follows:

`--Interface`

(1) r_0, r_1, w_0, w_1

`--Initial State`

(2) In the initial state the three bits contain the same value.
$c_0 \leftrightarrow c_1 \wedge c_0 \leftrightarrow c_2$

`--Specification`

(3) A safety property of the system: the three bits should coincide.
$\mathsf{OG}((c_0 \wedge c_1 \wedge c_2) \vee (\neg c_0 \wedge \neg c_1 \wedge \neg c_2))$

(4) The value read from the cell ought to coincide with the last writing performed.
$\mathsf{OG}((r_0 \rightarrow w_0) \wedge (r_1 \rightarrow w_1))$

(5) If a zero has been written, then w_1 is false and vice versa.
$\mathsf{AG}(w_0 \equiv \neg w_1)$

(6) Variable w_1 only changes when w_0 becomes true, and vice versa.
$\mathsf{AG}((w_0 \,\mathcal{U}\, w_1) \wedge (w_1 \,\mathcal{U}\, w_0))$

(7) The reading of a 0 corresponds to the value read by the majority.
$\mathsf{AG}(r_0 \equiv (\neg c_0 \wedge \neg c_1) \vee (\neg c_0 \wedge \neg c_2) \vee (\neg c_1 \wedge \neg c_2))$

(8) The reading of a 1 corresponds to the value read by the majority.
$\mathsf{AG}(r_1 \equiv (c_0 \wedge c_1) \vee (c_0 \wedge c_2) \vee (c_1 \wedge c_2))$

(9) If the user wants to write 1, then in the next step the memory will be setup to one.
$\mathsf{AG}(v = 1 \rightarrow \mathsf{AX}(w_1 \wedge v = \bot \wedge c_0 \wedge c_1 \wedge c_2))$

(10) Similar to the previous, but for 0.
$\mathsf{AG}(v = 0 \rightarrow \mathsf{AX}(w_0 \wedge v = \bot \wedge \neg c_0 \wedge \neg c_1 \wedge \neg c_2))$

(11) At any moment the user may decide to write a value.
$\mathsf{AG}(v = \bot \rightarrow \mathsf{AX}(v = 1 \vee v = 0 \vee v = \bot))$

Besides these formulas, one may add additional constraints, e.g., indicating that atomic steps (including faults) change bits by one. These constraints are straightforward to capture in CTL. Finally, the type of fault-tolerance we require is *masking fault-tolerance*.

Let us now illustrate how our synthesis approach works on this example. Figure 5.1 shows the partial tableau generated by Algorithm 6 for this problem. AND-nodes and OR-nodes are shown as rectangles and hexagons, respectively. For the sake of brevity, we put only the relevant information inside each box. Initially, a tableau is built using Algorithm 6, employing the rules α and β for CTL and dCTL- formulas until every node in the tableau has at least one successor. The tableau contains a fault injection part, generated from the AND-node in the second level of the tableau. This FOR-node is labeled identically as its AND-node predecessor. From this FOR-node we generate all possible faults from deontic formula violations. In particular, this node has deontic propositional variables O_{c_0}, O_{c_1}, and O_{c_2}, expressing that c_0, c_1, and c_2 should be true there, which is the case in this node. Now, we start to consider those cases in which an obligation might be violated. Following Algorithm 5, we negate one-by-one these deontic propositional variables. We generated three faulty AND-nodes (for the sake of brevity, just two of them are drawn) from the FOR-node with similar information to it except for the new negated propositional variable. The first FAND-node successor introduces $\neg c_0$ violating O_{c_0} (say f_0). The second and third

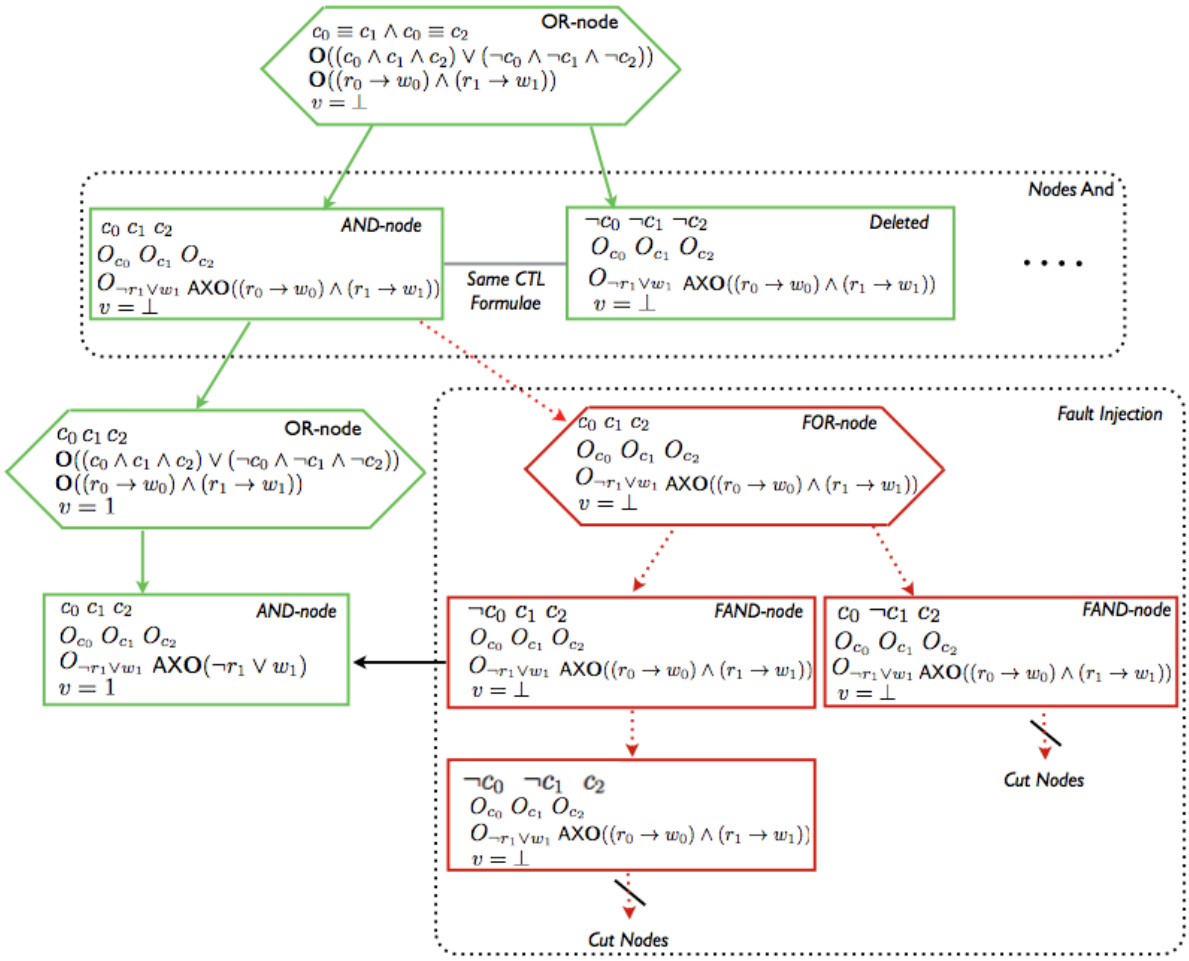

Figure 5.1: Partial tableau for a Memory Cell.

FAND-nodes introduce $\neg c_1$ and $\neg c_2$ violating O_{c_1} and O_{c_2}, respectively. Similarly for the other FAND-nodes. We continue the process of negating deontic propositional variables from these faulty AND-nodes. As a successor of f_0, we obtain the same information of f_0 with $\neg c_1$ (say f_0'). Thus, we have that O_{c_0} and O_{c_1} are violated. After the injection of faults process, we check whether it is possible to mask these faulty states using Algorithm 8. For example, for the case of the FAND-node which contains $\neg c_0$ and $\neg c_1$ (say f_0'), Algorithm 8 checks whether this FAND-node can be masked. Our algorithm cuts out this node because it cannot be masked. Similar

results are obtained for the other combinations. Moreover, for each masked FAND-node f, a (recovery) transition is added from it to each successor of $Mask(f)$ in the case that we can reach a normal successor using the rules of the tableau. Notice that faults introduced change a bit and keep the bits unchanged during the recovery process. After that, since all the faulty nodes that can be masked were generated, we check condition $B.2$ of the simulation relation by using Algorithm 9. This process may also cut out other faulty nodes: those which exhibit normal behavior which is not the behavior of the correct part of the system. Finally, we are ready to extract the fault-tolerant program from the tableau using the unfolding process (see Section 4.5). Figure 5.2 shows the transition diagram of the program extracted from the structure in Figure 5.1. For the sake of simplicity, the program does not include all the masked faults (these are similar to those shown in the program). Graphically, the normal states and transitions are represented as green rounded circles and lines, respectively. Faulty states and transitions are pictured as red rounded circles and lines, respectively. Recovery transitions are represented by black lines. The information contained in each state is interpreted as 000 (reading from left to right) for $c_0 = 0$, $c_1 = 0$, and $c_2 = 0$; moreover, the value of v is stated.

Note that this program was generated considering that faults are computed from deontic operators automatically, only considering some basic operations on the data structures of the states (in this case bits). Hence, we did not include the fault specification $fSpec$ for this problem. Other approaches [Kulkarni and Arora, 2000; Kulkarni and Ebnenasir, 2004; Attie et al., 2004] require faults to be given as input of the synthesis process, e.g., as special actions specified as guarded commands. We also allow to the users to specify the faults that can affect the normal behavior of the system via the fault specification $fSpec$. Therefore, we can add the following formula in the memory cell example:

(11) For $i = \{0, 1, 2\}$, at some point a bit may lose its charge.
$\mathsf{AG}(c_i \wedge v = \bot \rightarrow \mathsf{AX}(v = \bot \wedge \neg c_i))$

Notice that sentence (11) is covered in our synthesis process.

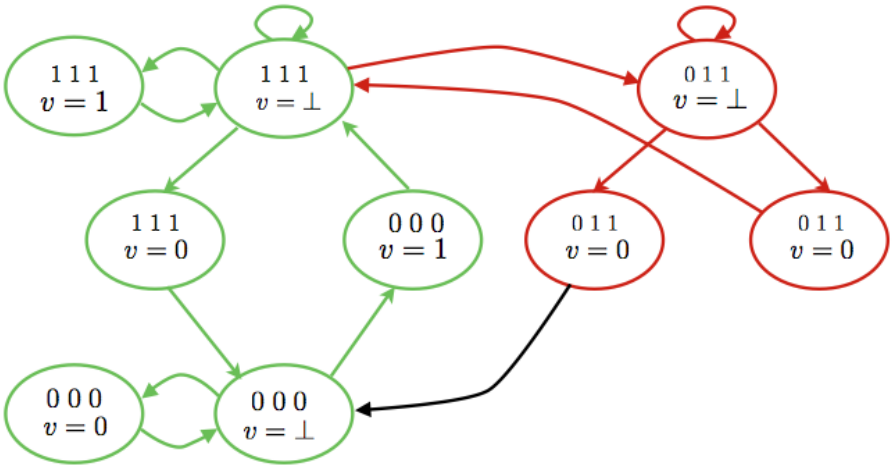

Figure 5.2: Part of the fault-tolerant program extracted from the structure in Figure 5.1.

5.2 Byzantine Agreement

Our second case study is the well-known Byzantine generals problem, introduced originally in [Lamport et al., 1982]. We have introduced the underlying idea of this example in Subsection 3.6.2.

We assume the following: G is the general, the messages are delivered correctly and all the lieutenants can communicate directly with each other; in this scenario they can recognize who is sending a message. Faults can convert loyal lieutenants into traitors (Byzantines). As a consequence, traitors might deliver false messages or perhaps they avoid sending a message that they received. The loyal lieutenants must agree on attacking or retreating after $m + 1$ rounds of communication, where m is the maximum numbers of traitors. The algorithm can ensure correct operation only if fewer than one third of the lieutenants are traitors. Finally, traitors cannot forge messages on behalf of loyal lieutenants.

We have formalized this problem for one general G and three lieutenants $L1$, $L2$, and $L3$. We have the following propositions:

- $G.a$, the general G wants to attack the enemy.
- $G.b$, indicates if the general G is Byzantine or not.
- $Li.b$ for $1 \leq i \leq 3$, states if lieutenant Li is Byzantine or not.

- $Li.Ga$ for $1 \leq i, j \leq 3$, indicates that lieutenant Li has received the message to attack from the general G.
- $Li.Lja$ for $1 \leq i, j \leq 3$, indicates that lieutenant Li has received the message to attack from lieutenant Lj.
- $Li.d$ for $1 \leq i \leq 3$, states that lieutenant Li has made a decision, either to attack or retreat.
- r_i for $0 \leq i \leq 1$, are propositions used to indicate the number of the current round of messages.
- $reset$, indicates that the system is reset to start a new round of messages.

Note that we only model the case when the general wants to attack; the case that he wants to retreat is symmetric to the one presented below.

The requirements on this system (*dSpec* and *interface*) can be specified in dCTL- as follows:

`--Interface`

(1) $L1.d$, $L2.d$, $L3.d$

`--Initial State`

(2) In the initial state, round r_0, the three lieutenants are not Byzantine and they did not make any decision yet. Moreover, the general G is not Byzantine and he has decided to attack.
$\neg L1.b \wedge \neg L2.b \wedge \neg L3.b \wedge G.a \wedge \neg G.b \wedge r_0$

`--Specification`

(3) In round 0 the general sends the message of attack to the lieutenants $L1$, $L2$, and $L3$.
$\mathsf{AG}((G.a \wedge r_0) \rightarrow \mathsf{AX}(r_1 \wedge L1.Ga \wedge L2.Ga \wedge L3.Ga \wedge G.a \wedge \neg L1.b \wedge \neg L2.b \wedge \neg L3.b))$

(4) Each non-Byzantine lieutenant forwards the message of attack to the other lieutenants in round 1.

(a) $\mathsf{AG}((L1.Ga \wedge \neg L1.b \wedge r_1) \rightarrow \mathsf{AX}(reset \wedge L2.L1a \wedge L3.L1a \wedge L1.Ga \wedge L2.Ga \wedge L3.Ga \wedge G.a \wedge \neg G.b \wedge \neg L1.b \wedge \neg L2.b \wedge \neg L3.b))$

(b) $\mathsf{AG}((L2.Ga \wedge \neg L2.b \wedge r_1) \rightarrow \mathsf{AX}(reset \wedge L1.L2a \wedge L3.L2a \wedge L1.Ga \wedge L2.Ga \wedge L3.Ga \wedge G.a \wedge \neg G.b \wedge \neg L1.b \wedge \neg L2.b \wedge \neg L3.b))$

(c) $\mathsf{AG}((L3.Ga \wedge \neg L3.b \wedge r_1) \rightarrow \mathsf{AX}(reset \wedge L1.L3a \wedge L2.L3a \wedge L1.Ga \wedge L2.Ga \wedge L3.Ga \wedge G.a \wedge \neg G.b \wedge \neg L1.b \wedge \neg L2.b \wedge \neg L3.b))$

(5) After the final round $m + 1$ is reached (m is the number of traitors or Byzantine allowed), the system is reset.
$\mathsf{AG}(reset \rightarrow \mathsf{AX}(\neg L1.b \wedge \neg L2.b \wedge \neg L3.b \wedge G.a \wedge \neg G.b \wedge r_0))$

(6) Each lieutenant L_j decides to attack if it has received at least two messages of attack coming from the other two lieutenants and the general.

(a) $\mathsf{AG}(L1.d \leftrightarrow ((L1.Ga \wedge L1.L2a \wedge L1.L3a) \vee (L1.Ga \wedge L1.L2a) \vee (L1.Ga \wedge L1.L3a)))$

(b) $\mathsf{AG}(L2.d \leftrightarrow ((L2.Ga \wedge L2.L1a \wedge L2.L3a) \vee (L2.Ga \wedge L2.L1a) \vee (L2.Ga \wedge L2.L3a)))$

(c) $\mathsf{AG}(L3.d \leftrightarrow ((L3.Ga \wedge L3.L1a \wedge L3.L2a) \vee (L3.Ga \wedge L3.L1a) \vee (L3.Ga \wedge L3.L2a)))$

(7) The systems has been reset iff the general has taken his decision.
$\mathsf{AG}(reset \leftrightarrow G.d)$

(8) A safety property of the system: the three lieutenants are not Byzantine (or traitors).
$\mathsf{OG}(\neg L1.b \wedge \neg L2.b \wedge \neg L3.b)$

(9) Validity: if the general G is not Byzantine, then the final decision of the lieutenants must be the same as that of the general G.
$\mathsf{OG}((\neg G.b \wedge reset) \rightarrow (G.d \wedge L1.d \wedge L2.d \wedge L3.d))$

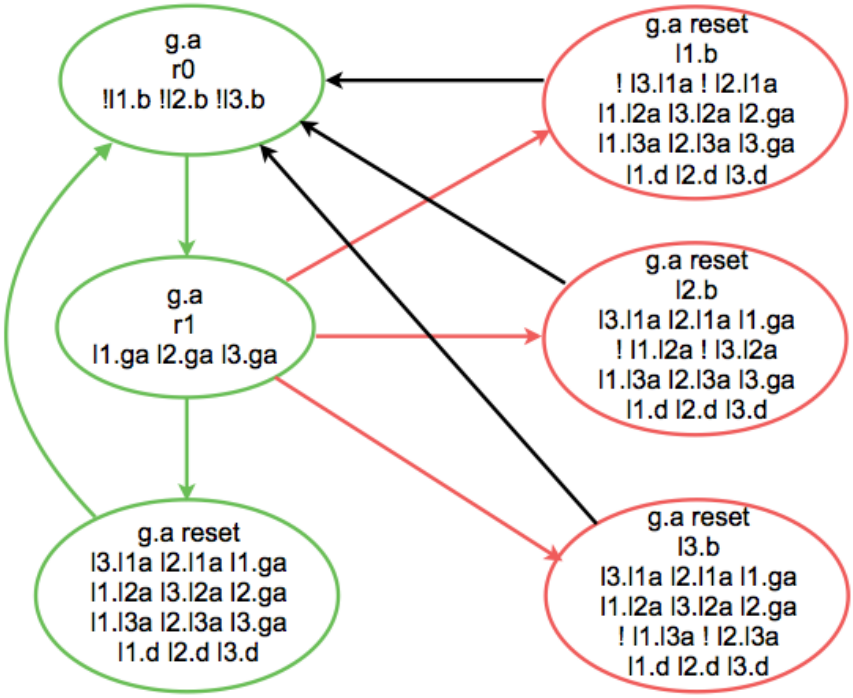

Figure 5.3: Fault-tolerant model synthesized for the Byzantine agreement problem.

(10) Agreement: the final decision of any 2 non-Byzantine lieutenants must be equal.
$\mathsf{OG}(((\neg L1.b \wedge \neg L2.b) \rightarrow (L1.d \wedge L2.d)) \wedge ((\neg L1.b \wedge \neg L3.b) \rightarrow (L1.d \wedge L3.d)) \wedge$
$((\neg L2.b \wedge \neg L3.b) \rightarrow (L2.d \wedge L3.d)))$

(11) If a lieutenant is Byzantine, then it means that he forwards the message of not attacking to the other lieutenants.

(a) $\mathsf{AG}(L1.b \leftrightarrow (\neg L2.L1a \wedge \neg L3.L1a))$

(b) $\mathsf{AG}(L2.b \leftrightarrow (\neg L1.L2a \wedge \neg L3.L2a))$

(c) $\mathsf{AG}(L3.b \leftrightarrow (\neg L1.L3a \wedge \neg L2.L3a))$

Finally, the type of fault-tolerance required is *masking fault-tolerance*.

The final masking fault-tolerant model synthesized using our technique is depicted in Figure 5.3. Note that in the resulting masking fault-tolerant model, after that the general G has sent the order to attack to all the lieutenants, only one of these could become a traitor. The cases in which at least two lieutenants become traitors are not

included in the model because these cannot be masked. On the other hand, the cases in which only one of the lieutenants is Byzantine are masked due to the agreement process.

5.3 N-Modular-Redundancy (NMR)

N-modular redundancy consist of N systems, in which these perform a process and that results are processed by a majority-voting system to produce a single output. An NMR system can tolerate up to n module failures, where $n = (N-1)/2$. For this case study, we have synthesized a 5-modular-redundancy example using our synthesis approach. The vocabulary of the example is given by the following set of propositions with their intuitive meaning:

- $pi.in$ for $1 \leq i \leq 5$, represents the input value for each process i.
- $input\_0$, the inputs for all processes have been set to 0.
- $input\_1$, the inputs for all processes have been set to 1.
- out_0, the output of the majority voting system is 0.
- out_1, the output of the majority voting system is 1.
- set_input, variable used to initiate a new cycle for setting the inputs of the modules.
- set_input_0 (set_input_1), variable used to simulate that the modules will receive as input the value 0 (1).

The requirements on this system (*dSpec* and *interface*) can be specified in dCTL- as follows:

```
--Interface
```

(1) $input\_0, input\_1, out\_0, out\_1$;

`--Initial State`

(2) Initial State: the input for the five modules are set to the same value.
$((\neg p1.in \wedge \neg p2.in \wedge \neg p3.in \wedge \neg p4.in \wedge \neg p5.in \wedge input\_0 \wedge set\_input) \vee$
$(p1.in \wedge p2.in \wedge p3.in \wedge p4.in \wedge p5.in \wedge input\_1 \wedge set\_input))$

`--Specification`

(3) A safety property of the system: the input for the five modules should coincide.
$\mathsf{OG}((p1.in \wedge p2.in \wedge p3.in \wedge p4.in \wedge p5.in) \vee$
$(\neg p1.in \wedge \neg p2.in \wedge \neg p3.in \wedge \neg p4.in \wedge \neg p5.in))$

(4) The value of the output ought to coincide with the last input.
$\mathsf{OG}((\neg out \rightarrow input\_0) \vee (out \rightarrow input\_1))$

(5) The value of the output out ought to coincide with the value obtained from the majority gate, at most 2 faults are tolerated.

(a) $\mathsf{AG}(out\_1 \leftrightarrow ((p1.in \wedge p2.in \wedge p3.in) \vee (p1.in \wedge p2.in \wedge p4.in) \vee$
$(p1.in \wedge p3.in \wedge p4.in) \vee (p2.in \wedge p3.in \wedge p4.in) \vee$
$(p2.in \wedge p3.in \wedge p5.in) \vee (p2.in \wedge p4.in \wedge p5.in) \vee$
$(p3.in \wedge p4.in \wedge p5.in) \vee (p1.in \wedge p4.in \wedge p5.in) \vee$
$(p1.in \wedge p2.in \wedge p5.in) \vee (p1.in \wedge p3.in \wedge p5.in)))$

(b) $\mathsf{AG}(out\_0 \leftrightarrow ((\neg p1.in \wedge \neg p2.in \wedge \neg p3.in) \vee (\neg p1.in \wedge \neg p2.in \wedge \neg p4.in) \vee$
$(\neg p1.in \wedge \neg p3.in \wedge \neg p4.in) \vee (\neg p2.in \wedge \neg p3.in \wedge \neg p4.in) \vee$
$(\neg p2.in \wedge \neg p3.in \wedge \neg p5.in) \vee (\neg p2.in \wedge \neg p4.in \wedge \neg p5.in) \vee$
$(\neg p3.in \wedge \neg p4.in \wedge \neg p5.in) \vee (\neg p1.in \wedge \neg p4.in \wedge \neg p5.in) \vee$
$(\neg p1.in \wedge \neg p2.in \wedge \neg p5.in) \vee (\neg p1.in \wedge \neg p3.in \wedge \neg p5.in)))$

(6) If the input is intended to be set to 1, then in the next step the input for the five modules will be set up to one.
$\mathsf{AG}(set\_input\_1 \rightarrow \mathsf{AX}(p1.in \wedge p2.in \wedge p3.in \wedge p4.in \wedge p5.in \wedge input\_1 \wedge set\_input))$

(7) If the input is intended to be set to 0, then in the next step the input for the five modules will be set to zero.
$\mathsf{AG}(set\_input\_0 \rightarrow \mathsf{AX}(\neg p1.in \wedge \neg p2.in \wedge \neg p3.in \wedge \neg p4.in \wedge \neg p5.in \wedge input\_0 \wedge$
$set\_input))$

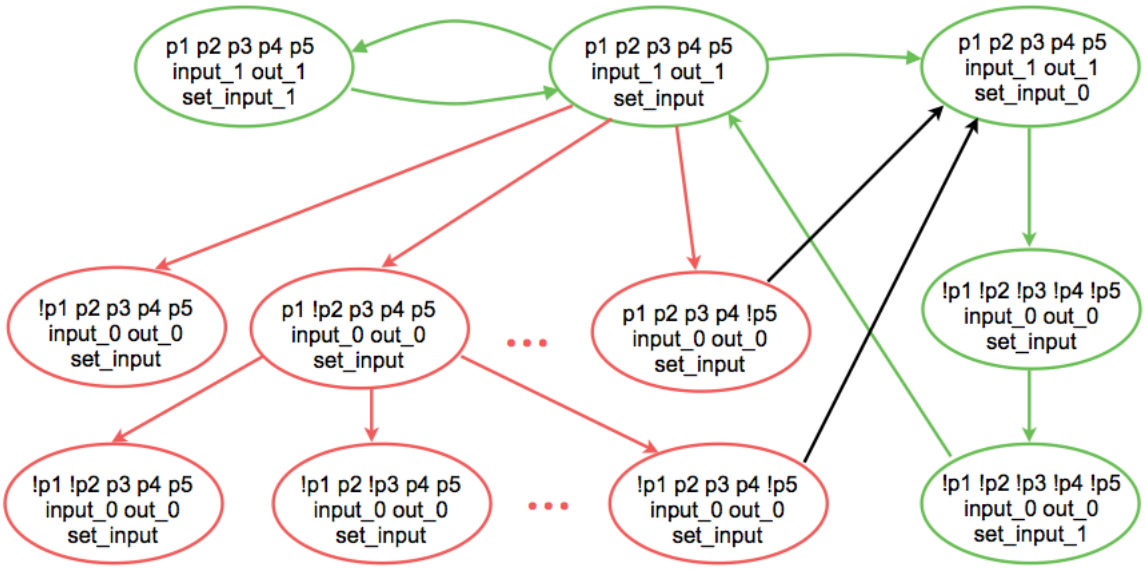

Figure 5.4: Part of the fault-tolerant program synthesized for the 5MR.

(8) At any moment the input for the five modules can be set to 0 or 1.

(a) $\mathsf{AG}((input\_0 \wedge set\_input \wedge \neg p1.in \wedge \neg p2.in \wedge \neg p3.in \wedge \neg p4.in \wedge \neg p5.in) \rightarrow \mathsf{AX}((set\_input\_0 \wedge input\_0) \vee (set\_input\_1 \wedge input\_0) \vee (set\_input \wedge input\_0)))$

(b) $\mathsf{AG}((input\_1 \wedge set\_input \wedge p1.in \wedge p2.in \wedge p3.in \wedge p4.in \wedge p5.in) \rightarrow \mathsf{AX}((set\_input\_0 \wedge input\_1) \vee (set\_input\_1 \wedge input\_1) \vee (set\_input \wedge input\_1)))$

Finally, the type of fault-tolerance required is *masking fault-tolerance*.

Part of the masking fault-tolerant implementation synthesized for the 5-modular-redundancy (5MR) problem is illustrated in Figure 5.4.

The resulting model contains many faulty states which are masked. We only include some of them, originating from one of the normal states. Note that, from this state, the 5MR can tolerate up to 2 module failures. Using our method, we have injected all possible faults for the five modules and we have pruned those faults that contain more than 2 module failures.

5.4 Token Ring

Our fourth example involves an adaptation of a case study from [Bonakdarpour et al., 2012], a token ring for solving distributed mutual exclusion, where processes $0 \dots N$ are organized in a ring and the token is circulated along the ring in a fixed direction. Each process, say p where p is in $\{0 \dots n\}$, maintains a variable $p_i.t$ with domain $\{0, 1\}$. Moreover, $pi.corr$ denotes that process pi contains a corrupted value. Process p, $0 \leq p \leq n-1$, has the token and can access the critical section iff $p_i.t$ differs from its successor $p_{i+1}.t$ and process n has the token iff $p_n.t$ is the same as its successor $p_0.t$. We have formalized the behavior of this system for four processes $p0, p1, p2$, and $p3$. Additionally, we have the proposition $pi.token$ for each process, which denotes that process pi has the token. These propositions will conform to the visible part of the system.

The requirements on this system ($dSpec$ and $interface$) can be specified in dCTL- as follows:

```
--Interface
```

(1) $p0.token, p1.token, p2.token, p3.token$

```
--Initial State
```

(2) Initial State: process $p0$ has the token and the four processes have not corrupted values.
$((p0.t \wedge p1.t \wedge p2.t \wedge p3.t) \vee (\neg p0.t \wedge \neg p1.t \wedge \neg p2.t \wedge \neg p3.t)) \wedge$
$(\neg p0.corr \wedge \neg p1.corr \wedge \neg p2.corr \wedge \neg p3.corr)$

```
--Specification
```

(3) A safety property of the system: there is no any process with a corrupted value.
$\mathsf{OG}(\neg p0.corr \wedge \neg p1.corr \wedge \neg p2.corr \wedge \neg p3.corr)$

(4) There is always exactly one node which has the token.
$\mathsf{OG}((p0.token \wedge \neg p1.token \wedge \neg p2.token \wedge \neg p3.token) \vee$
$(\neg p0.token \wedge p1.token \wedge \neg p2.token \wedge \neg p3.token) \vee$
$(\neg p0.token \wedge \neg p1.token \wedge p2.token \wedge \neg p3.token) \vee$
$(\neg p0.token \wedge \neg p1.token \wedge \neg p2.token \wedge p3.token))$

(5) Whenever a node has a token, it eventually passes it to the next one in the ring.
$\mathsf{OG}(\ (p0.token \rightarrow \mathsf{AX}(p1.token)) \wedge (p1.token \rightarrow \mathsf{AX}(p2.token)) \wedge$
$(p2.token \rightarrow \mathsf{AX}(p3.token)) \wedge (p3.token \rightarrow \mathsf{AX}(p0.token))\ )$

(6) Process p_i has the token iff p_i is not corrupted and $p_i.t$ differs from its successor $p_{i+1}.t$ and process n has the token iff $p_n.t$ is the same as its successor $p_0.t$.

(a) $\mathsf{AG}(p0.token \leftrightarrow (\ ((p0.t \wedge p1.t \wedge p2.t \wedge p3.t) \vee (\neg p0.t \wedge \neg p1.t \wedge \neg p2.t \wedge \neg p3.t)) \wedge$
$(\neg p0.corr \wedge \neg p1.corr \wedge \neg p2.corr \wedge \neg p3.corr))\ )$

(b) $\mathsf{AG}(p1.token \leftrightarrow (\ ((\neg p0.t \wedge p1.t \wedge p2.t \wedge p3.t) \vee (p0.t \wedge \neg p1.t \wedge \neg p2.t \wedge \neg p3.t)) \wedge$
$(\neg p0.corr \wedge \neg p1.corr \wedge \neg p2.corr \wedge \neg p3.corr))\ )$

(c) $\mathsf{AG}(p2.token \leftrightarrow (\ ((\neg p0.t \wedge \neg p1.t \wedge p2.t \wedge p3.t) \vee (p0.t \wedge p1.t \wedge \neg p2.t \wedge \neg p3.t)) \wedge$
$(\neg p0.corr \wedge \neg p1.corr \wedge \neg p2.corr \wedge \neg p3.corr))\ )$

(d) $\mathsf{AG}(p3.token \leftrightarrow (\ ((\neg p0.t \wedge \neg p1.t \wedge \neg p2.t \wedge p3.t) \vee (p0.t \wedge p1.t \wedge p2.t \wedge \neg p3.t)) \wedge$
$(\neg p0.corr \wedge \neg p1.corr \wedge \neg p2.corr \wedge \neg p3.corr))\ )$

(7) Id process p_0 has the token, then in the next state the token is passed to p_1 and there is no process with a corrupted value.

(a) $\mathsf{AG}((p0.t \wedge p1.t \wedge p2.t \wedge p3.t) \rightarrow \mathsf{AX}(\neg p0.t \wedge p1.t \wedge p2.t \wedge p3.t)\ )$

(b) $\mathsf{AG}((\neg p0.t \wedge \neg p1.t \wedge \neg p2.t \wedge \neg p3.t) \rightarrow \mathsf{AX}(p0.t \wedge \neg p1.t \wedge \neg p2.t \wedge \neg p3.t)\ )$

(8) Id process p_1 has the token, then in the next state the token is passed to p_2 and there is no process with a corrupted value.

(a) $\mathsf{AG}((\neg p0.t \wedge p1.t \wedge p2.t \wedge p3.t) \rightarrow \mathsf{AX}(\neg p0.t \wedge \neg p1.t \wedge p2.t \wedge p3.t)\ )$

(b) $\mathsf{AG}((p0.t \wedge \neg p1.t \wedge \neg p2.t \wedge \neg p3.t) \rightarrow \mathsf{AX}(p0.t \wedge p1.t \wedge \neg p2.t \wedge \neg p3.t)\ )$

(9) Id process p_2 has the token, then in the next state the token is passed to p_3 and there is no process with a corrupted value.

(a) $\mathsf{AG}((\neg p0.t \wedge \neg p1.t \wedge p2.t \wedge p3.t) \rightarrow \mathsf{AX}(\neg p0.t \wedge \neg p1.t \wedge \neg p2.t \wedge p3.t)\ )$

(b) $\mathsf{AG}((p0.t \wedge p1.t \wedge \neg p2.t \wedge \neg p3.t) \rightarrow \mathsf{AX}(p0.t \wedge p1.t \wedge p2.t \wedge \neg p3.t)\ )$

(10) Id process p_3 has the token, then in the next state the token is passed to p_0 and there is no process with a corrupted value.

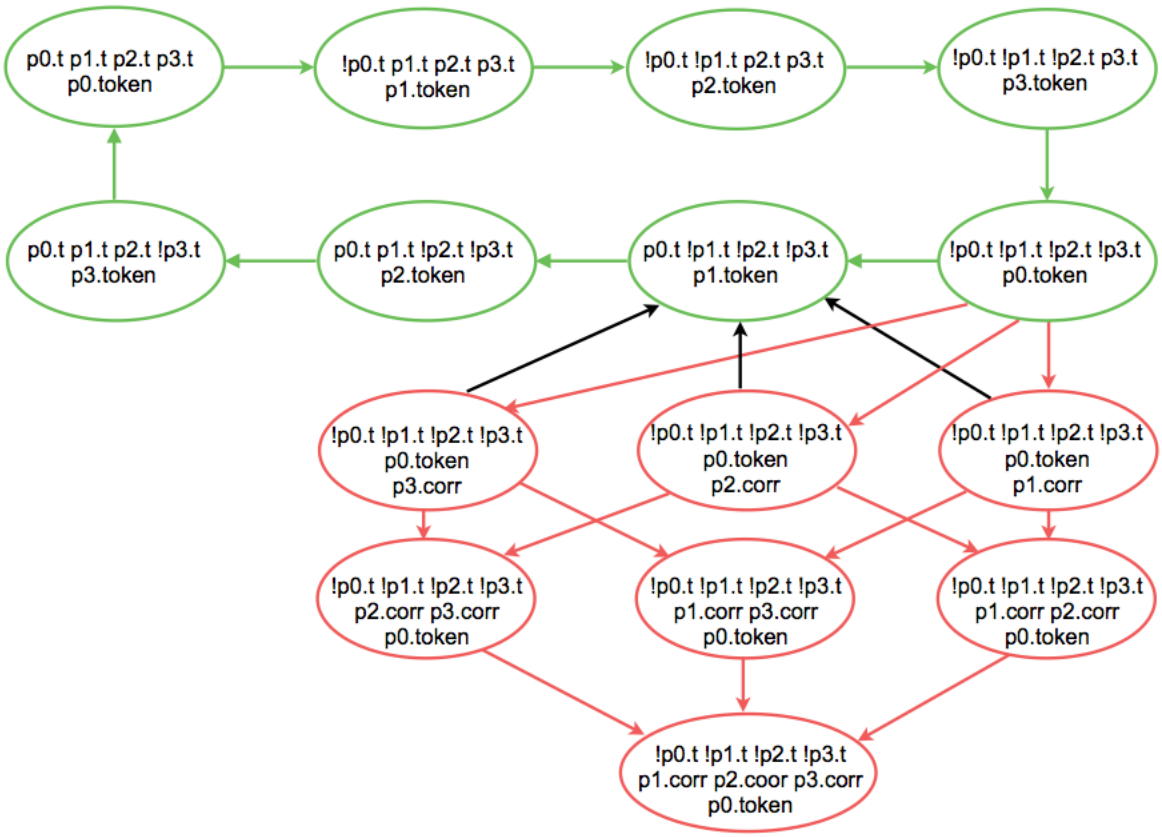

Figure 5.5: Part of the fault-tolerant program synthesized for the token ring.

(a) $\mathsf{AG}((\neg p0.t \wedge \neg p1.t \wedge \neg p2.t \wedge p3.t) \rightarrow \mathsf{AX}(\neg p0.t \wedge \neg p1.t \wedge \neg p2.t \wedge \neg p3.t)\ )$

(b) $\mathsf{AG}((p0.t \wedge p1.t \wedge p2.t \wedge \neg p3.t) \rightarrow \mathsf{AX}(p0.t \wedge p1.t \wedge p2.t \wedge p3.t)\ )$

Finally, the level of fault-tolerance required is *masking fault-tolerance*.

The synthesized masking fault-tolerant implementation for this problem is illustrated in Figure 5.5. Note that in all normal (green) states, the four processes do not have corrupted values, i.e., in each normal state $\neg p0.corr$, $\neg p2.corr$, $\neg p3.corr$, and $\neg p4.corr$ hold. Moreover, we have included in this model only the masked faults originating from one normal state, namely the case when $p1$ has the token. For the other cases, the result is similar to that presented in the model.

5.5 The Muller C-element with a majority circuit

We now present our fifth case study, the Muller C-element with a majority circuit. We have introduced it in Subsection 3.6.1, where we have stated the logical behavior of the system. In summary, it is composed of three boolean inputs x, y, and u and one boolean output z. We have the predicate $maj(x, y, u)$ which returns the value of the majority circuit, which is assumed to work correctly, and is defined as $maj(x, y, u) = (x \wedge y) \vee (x \wedge u) \vee (y \wedge u)$.

We specify in dCTL- the requirements of this system (*dSpec* and *interface*) as follows:

```
--Interface
```

(1) x, y, u, z

```
--Initial State
```

(2) Initial State: the three inputs and the output z contain the same value.
$(x \wedge y \wedge u \wedge z) \vee (\neg x \wedge \neg y \wedge \neg u \wedge \neg z)$

```
--Specification
```

(3) The two inputs x and y should coincide.
$\mathsf{OG}(x \leftrightarrow y)$

(4) The values of u and z should coincide.
$\mathsf{OG}(u \leftrightarrow z)$

(5) maj contains the value of the majority circuit over x, y, and u, which is assumed to work correctly.
$\mathsf{AG}(maj \equiv (x \leftrightarrow y) \vee (x \leftrightarrow u) \vee (y \leftrightarrow u))$

(6) Input x (respectively, y) changes only if $x = z$ (respectively, $y = z$), i.e, x and y change simultaneously.

(a) $\mathsf{AG}((x \wedge y \wedge z) \rightarrow \mathsf{AX}(\neg x \wedge \neg y \wedge z))$

(b) $\mathsf{AG}((\neg x \wedge \neg y \wedge \neg z) \rightarrow \mathsf{AX}(x \wedge y \wedge \neg z))$

(7) u and z change simultaneously.

(a) $\mathsf{AG}((\neg maj \wedge u \wedge z) \rightarrow \mathsf{AX}(\neg maj \wedge \neg u \wedge \neg z))$

(b) $\mathsf{AG}((maj \wedge \neg u \wedge \neg z) \rightarrow \mathsf{AX}(maj \wedge u \wedge z))$

Finally, the level of fault-tolerance required is *nonmasking fault-tolerance*.

Part of the nonmasking fault-tolerant implementation synthesized for the Muller C-element with majority circuit is illustrated in Figure 5.6. We have defined as *interface* the four variables (x, y, u, and z) used in this problem, i.e., we focus on these variables as the visible part of system to analyse the required level of fault-tolerance. Notice that deontic violations are produced from statements 3 and 4, which are all the possible faults that can be generated from this specification. Intuitively, the faults illustrated in Figure 5.6 cannot be masked, but there exists a nonmasking relation where eventually a normal state is reached from any of the faulty states. We remark that, the fault-tolerant program synthesized in Figure 5.6 coincide with the one presented in Figure 3.5.

On the other hand, if we only consider faults that corrupt the values of the input x and y, i.e., we need to define as *interface* of this problem the variables u and z, then we can synthesize a masking fault-tolerant version of Muller C-element with majority circuit. Part of the resulting model, using syntdctl, is depicted in Figure 5.7. Observe in the model that faults only affect variables x and y, generated by the negation of deontic variables from formula (3). Because this systems is composed of a majority circuit, these faults are masked.

5.6 Altitude Switch (ASW)

Our sixth case study involves an adaptation of the Altitude Switch (ASW) controller presented in [Jeffords et al., 2009]. We have introduced this example in Subsection 3.6.3. In short, the ASW controller is responsible for turning on a Device of Interest (DOI) when the aircraft altitude is below a pre-specified threshold. Essentially, the ASW controller reads a set of variables and produces an output.

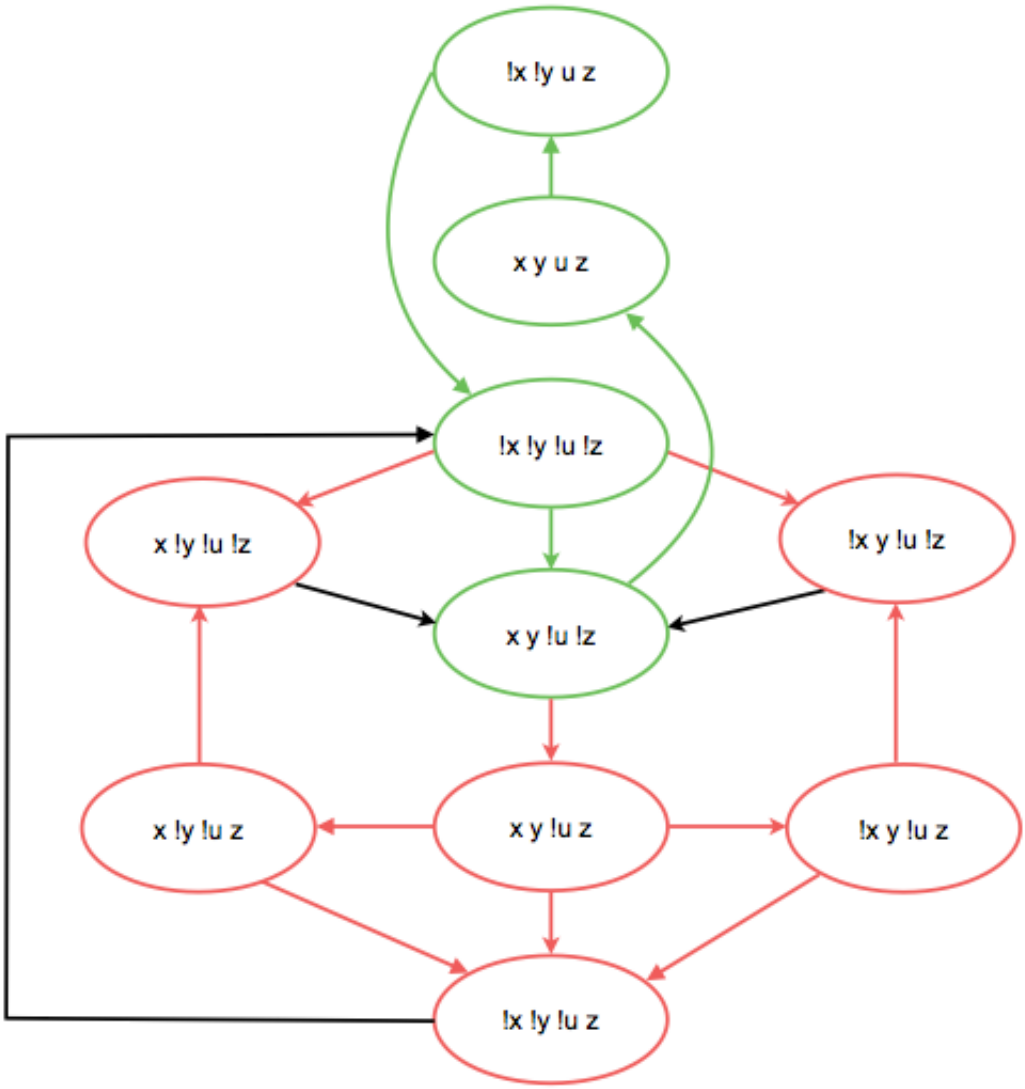

Figure 5.6: Part of the nonmasking fault-tolerant program synthesized for the Muller C-element.

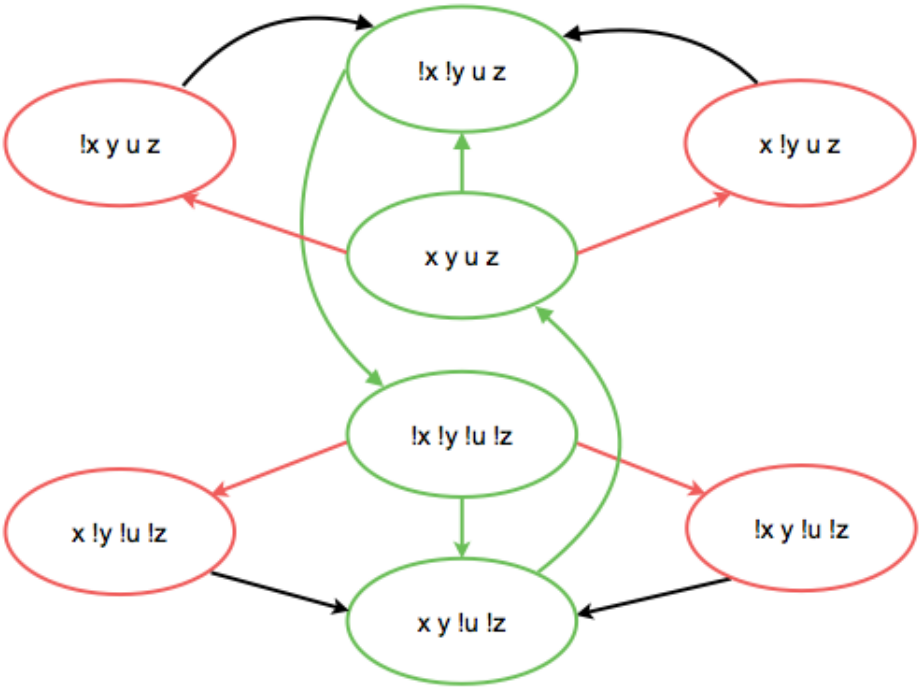

Figure 5.7: Part of the masking fault-tolerant program synthesized for the Muller C-element.

The vocabulary of the example is given by the following set of propositions for specifying the normal behavior of the system with their intuitive meaning:

- *complete_init*, true if the initialization of the system is complete,
- *alt_below*, true if the altitude is below a pre-specific threshold,
- *doi_status*, true if the DOI is powered on,
- *inhibit*, true when the DOI power-on is inhibited,
- *reset*, true if the system is being reset.

Additionally, the ASW system can be subject to hardware malfunctions that may alter the ASW controller. The following are the propositions used to represent three time-out potential faults, which will be used for describing the fault specification of this problem:

- $init\_fail$, true if the initialization fails,
- alt_fail, true if the altimeter fails,
- doi_fail, true if the device of inters fails.

Finally, we define the next propositions for expressing the different modes that the ASW controller can be:

- *init*, true if the ASW system is initializing,
- *await_doi*, true if the system is waiting for the DOI to power on,
- *stanby*, true for all other cases,
- fm, true when the system detects any of the above faults.

The requirements on this system (*dSpec* and *interface*) can be specified in dCTL- as follows:

--Interface

(1) $init, standby, await\_doi, fm$

--Initial State

(2) Initial State: the system is in the mode *init*, i.e., initializing the ASW controller.
$init \wedge \neg inhibit \wedge \neg doi\_status$

--Specification

(3) Once the initialization of the system is complete, then the system moves to the standby mode.
$\mathsf{AG}((init \wedge complete\_init) \rightarrow \mathsf{AX}(standby))$

(4) The system moves from standby mode to Await DOI.
$\mathsf{AG}((standby \wedge \neg inhibit \wedge \neg doi\_status) \rightarrow \mathsf{AX}(await\_doi))$

(5) The system moves from Await DOI to standby.
$\mathsf{AG}(await\_doi \rightarrow \mathsf{AX}(standby))$

(6) The system is at in most one of the modes *init*, *standby*, or *await_doi*.
$\mathsf{OG}(init \rightarrow \neg(standby \vee await\_doi)) \wedge \mathsf{OG}(standby \rightarrow \neg(init \vee await\_doi)) \wedge \mathsf{OG}(await\_doi \rightarrow \neg(init \vee standby))$

--Fault Specification

(7) The system in the faulty mode is not in any of the modes *init*, *standby*, or *await_doi*.
$\mathsf{AG}(fm \equiv (\neg init \vee \neg standby \vee \neg await\_doi))$

(8) The system detects that initialization of the ASW controller fails and it goes into the faulty mode fm.
$\mathsf{AG}((init \wedge init\_fail) \rightarrow \mathsf{AX}(fm))$

(9) The system detects that the altimeter failed and it goes into the faulty mode fm.
$\mathsf{AG}((standby \wedge alt\_fail) \rightarrow \mathsf{AX}(fm))$

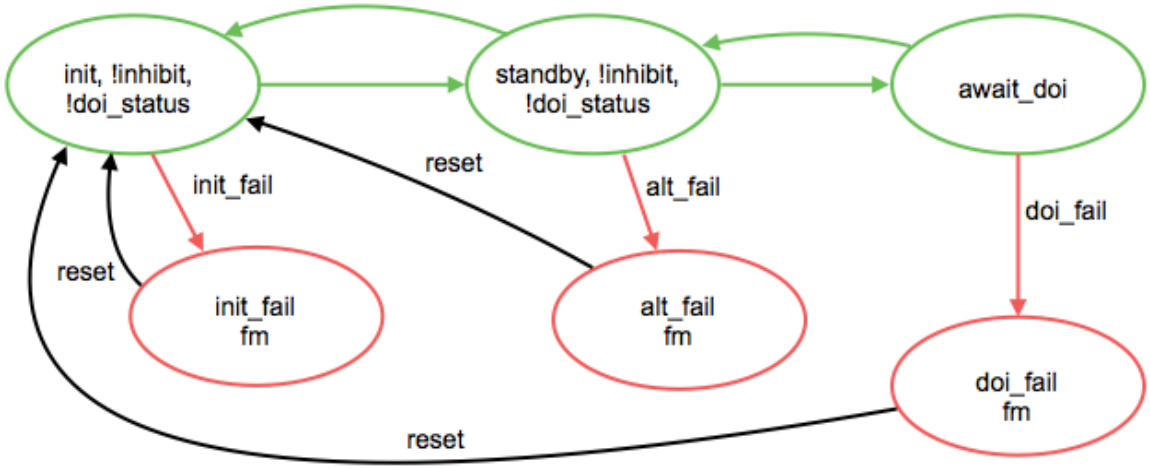

Figure 5.8: Part of the nonmasking fault-tolerant program synthesized for the ASW controller.

(10) The system detects that the device of interest failed and it goes into the faulty mode fm.
$\mathsf{AG}((await\_doi \wedge doi\_fail) \rightarrow \mathsf{AX}(fm))$

(11) The system can recover from the faulty state fm, if the system has been reset by the pilot.
$\mathsf{AG}((fm \wedge reset) \rightarrow \mathsf{AX}(init))$

The level of fault-tolerance required is *nonmasking fault-tolerance*.

The synthesized nonmasking fault-tolerant implementation for this problem is illustrated in Figure 5.8. The result obtained here is similar to the one presented in Figure 3.9, where the last one is an abstraction of the presented in Figure 5.8. In more details, there are three faulty states depicted in Figure 3.9 which represent the initialization fails, the altimeter fails, and device of inters fails, respectively. Once any of these faults occurs, the system goes into the faulty mode fm. On the other hand, we have represented these three faulty states with only one faulty state labeled with the faulty mode fm in Figure 5.8.

This is the first case study where we have used the option of the fault specification

specified from statement (7) to (11). Essentially, we represent in these assertions three potential time-out potential faults and how the system recovers to a normal state of the system. We must remark that we assume that the system detects these faults and then these propositional variables become true in each case, and similarly for the case when the pilot presses the reset button.

Finally, we have obtained results similar to that of [Jeffords et al., 2009] and [Abujarad and Kulkarni, 2008], where in the later they have added fault-tolerance automatically to the requirements specification of the Altitude Switch controller described with the SCR (Software Cost Reduction) formal method.

5.7 A Simple Train System

Our final case study is a Simple Train System, which we have introduced in Subsection 3.6.4. In summary, the system in general consists of n trains and m rail segments. Rail segments are connected to other rail segments, where in each of these connections the rails are equipped with a signal which indicates if the segment is occupied or not. The signals can be green (when the segment is free) or red (when a train is in the segment).

We consider a model defining variables for three trains t_i with $0 \leq i \leq 2$ and five rail segments r_j with $0 \leq j \leq 4$, where they are connected as follows: $r_0 R r_1$, $r_1 R r_2$, $r_2 R r_3$, and $r_2 R r_4$. In addition, we have the following propositions:

- $t_i.stop$ denotes that train t_i is stopped,
- $r_i R r_j$ means that segments i and j are connected,
- $r_j.green$ expresses that the signal of segment j is green,
- $r_j.red$ expresses that the signal of segment j is red.

The requirements on this system (*dSpec* and *interface*) can be specified in dCTL- as follows:

```
--Interface
```

(1) r_0, r_1, r_2, r_3, r_4

`--Initial State`

(2) The initial state starts with a specific configuration of the trains over the rail segments.
$t_0.r_0 \wedge t_1.r_2 \wedge t_2.r_3 \wedge r_0.red \wedge r_1.green \wedge r_2.red \wedge r_3.redr_4.green \wedge r_0Rr_1 \wedge r_1Rr_2 \wedge r_2Rr_3 \wedge r_2Rr_4$

`--Specification`

(3) There are no two trains in the same segment (for $0 \leq i, j \leq 2$ and $0 \leq k \leq 4$).
$\mathsf{OG}(\neg(t_i.r_k \wedge t_j.r_k))$

(4) When there is a train t_i in the rail segment r_j, then the signal should be red for this segment.
$\mathsf{OG}(t_i.r_j \rightarrow r_j.red)$

(5) If there is no train in the segment r_j, then the signal must be green on this segment.
$\mathsf{OG}(\neg t_i.r_j \rightarrow r_j.green)$

(6) If the signal of a segment is red, then any train is forbidden to move into the segment.
$\mathsf{OG}((\neg r_j.green \wedge \neg t_i.r_j) \rightarrow \mathbf{F}(t_i.r_j))$

(7) Train t_k moves from rail r_i to r_j if both are connected and the signal on the rail r_j is green.
$\mathsf{AG}((t_k.r_i \wedge r_j.green \wedge r_iRr_j) \rightarrow \mathsf{AX}(t_k.r_j \wedge r_i.green \wedge r_j.red))$

`--Fault Specification`

(8) A communication fault of the system is detected, where the block instruments on each rail segment react to this malfunction, turning the signal to red.
$\mathsf{AG}(comun\_fail \rightarrow \mathsf{AX}(r_i.red))$

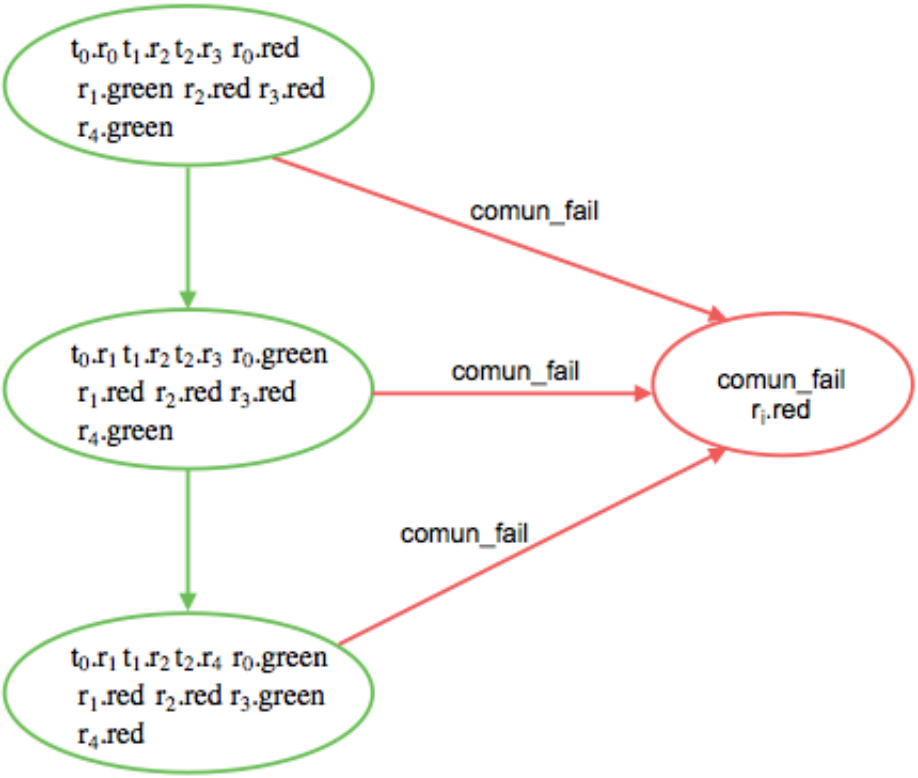

Figure 5.9: Part of the failsafe fault-tolerant program synthesized for the train system.

Finally, the level of fault-tolerance required is *failsafe fault-tolerance*.

The synthesized failsafe fault-tolerant implementation for this problem is illustrated in Figure 5.9. We want to remark that in all normal (green) states the rail segments are connected as follow: r_0Rr_1, r_1Rr_2, r_2Rr_3, and r_2Rr_4, where we did not include these propositions in the figure for reasons of space. Notice that we have introduced a fault specification for this problem in assertion (8), which essentially injects faults when a malfunction in the communication of the system is detected and then the signals of all rail segments are turned to red. Additionally, we have also injected faults through the violation of deontic variables from statement (3) to (6), but none can be masked. For example, violating assertion (3) leads to the case that two trains are in the same rail segment; consequently, a train collision can result. Of course, this kind of fault is not at all desirable in this system.

On the other hand, the faults injected from statement 8 satisfy failsafe fault-tolerance because, at the faulty state of Figure 5.9, all the safety properties hold considering the visible part of the system (*interface*), i.e., the five rail segments. We must remark that we assume that the system detects this malfunction in the communication of the system, setting the propositional variable $comun\_fail$ to true. This assumption then becomes a requirement to be implemented by the system. Finally, the model synthesized in Figure 5.9 is the same to the one presented in Figure 3.10.

5.8 Description of the syntdctl tool

In this section we briefly describe the architecture of syntdctl and the experimental results for some of the above case studies when using this tool. We have implemented in our tool only the cases of masking and failsafe fault-tolerance.

syntdctl is free software. Documentation and installation instructions can be found at `https://code.google.com/p/synt-dctl/`.

5.8.1 Tool Architecture

The architecture of syntdctl is illustrated in Figure 5.10. The input of syntdctl is a deontic specification *dSpec*, composed of an *init-spec*, a *normal-spec*, an $interface$, and a fault specification $fSpec$. *interface* is described by a subset of the state variables, which intuitively constitutes the visible part of the system; *init-spec* and *normal-spec* are dCTL- formulas, where the former specifies the initial states of the system, and the latter specifies properties that are required to hold in all states that are reachable from the initial state. Finally, the fault specification $fSpec$, is made of a set of *fault-variables*, a *fault-spec*, and a *combine-spec*. *fault-variables* is a set of auxiliary atomic propositions, and *fault-spec* specifies the faulty and recovery behavior over the atomic propositions (including *fault-variables*), using CTL formulas. Finally, *combine-spec* are also CTL formulas which relate the atomic propositions in the deontic specification *dSpec* with those in the fault specification $fSpec$. We remark that $fSpec$ is optional for the user.

Initially, syntdctl reads a deontic specification *dSpec* as an input file, which is then tokenized (Lexer) and parsed to obtain abstract syntax trees according to the dCTL-

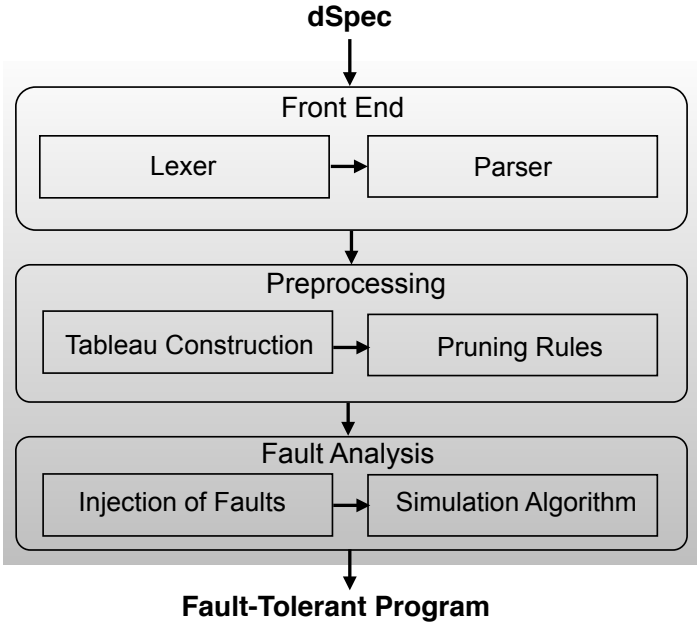

Figure 5.10: The Arquitecture of syntdctl.

expression grammar (Parser). The abstract syntax trees are stored as elements of a set of dCTL- formulas. The preprocessing component constructs an initial tableau T_N for the input $dSpec$ based on a dCTL- SAT procedure. Pruning rules are applied to the the tableau T_N in order to remove all nodes that are either propositionally inconsistent, do not have enough successors, or are labeled with an CTL or deontic eventuality formula which is not fulfilled. This process returns as a result $true$, if $dSpec$ is satisfiable, or $false$, in the case $dSpec$ is unsatisfiable. If $dSpec$ is satisfiable, it has a finite model that can be embedded in the tableau T_N. Assuming a positive result from the dCTL- decision procedure for $dSpec$, the next step is to perform a fault analysis. In this phase, faults are injected into the tableau in the first place, where faults are understood as (all possible) violations to the deontic obligations imposed in the description of the correct behavior of the system. Additionally, in case that the user has provided a fault specification $fSpec$ in the input file, we introduce the corresponding faults into the tableau. Subsequently, a masking or failsafe simulation algorithm (taking into account the input interface) is executed (depending on the desired user's level of fault-tolerance) in order to remove those nodes from the tableau that do not satisfy the required level of fault-tolerance. Finally, the tableau

T_F is unravelled into a masking or failsafe fault-tolerant program implementing $dSpec$.

Table 5.1: Experimental results.

Name	Faults Injected	faults unmasked/removed	Time in sec
Byzantine Agreement	7	4	0.20
Token Ring	220	150	111.85
5-Modular-Redundancy	410	260	535.91
Memory Cell	100	70	10.13
Muller C-Element	6	0	0.09
Simple Train System	28	25	2.3

Table 5.1 summarizes the experimental results on some of the models described above, reporting the number of faults injected and unmasked/removed to achieve masking or failsafe fault-tolerance, and corresponding running times.

The syntdctl tool is implemented in Java. In order to run the tool, Java 1.7+ is required. All experiments have been conducted on a computer with a 2.9 Ghz Intel Core i5 with 4 GB of memory.

Chapter 6

Concluding Remarks

In this dissertation, we focused on the problem of *automatically synthesizing fault-tolerant systems from logical specifications*. More specifically, the synthesis approach we have developed aims at automatically constructing a fault-tolerant component implementation from a logical specification of the component, and a user-specified required level of fault-tolerance (e.g., masking, nonmasking, failsafe). System specifications are provided in dCTL-, a branching time temporal logic equipped with further "deontic" modalities, that enable one to specify fault-tolerance related properties by distinguishing through these additional modalities the normal (non faulty) and the abnormal (involving faults) executions of the system. The synthesis technique is based on a combination of a tableau-based synthesis from CTL specifications, adapted and extended to deal with the additional modalities, and bisimulation algorithms, since different fault-tolerance levels are captured as particular (bi)simulation relations.

In this chapter, we present an overall picture of the status of our research on this problem. In Section 6.1, we discuss the related work in the literature of automated synthesis of fault-tolerant programs. Moreover, we summarize in Section 6.2 the contributions made in this dissertation. Finally, in Section 6.3, we describe some further work that seems interesting to develop using the framework described in the preceding chapters.

6.1 Related Work

The first main contribution of our work presented in Chapter 3 is closely related to several formal approaches to fault-tolerance. The programming style we use here for describing our programs was introduced in [Arora and Gouda, 1993; Arora and Kulkarni, 1998a,b; Gärtner, 1999a], where programs are written using Dijkstra's guarded command language, and faults are specified as distinguished actions. In these works, the authors characterize fault-tolerance using sets of states, for instance, a set P of states captures the desired invariant of the program, whereas a set T is employed to indicate those states that tolerate faults. In this setting, masking, nonmasking and failsafe fault-tolerance are formalized making use of liveness and safety properties, where properties in general are written using first-order logic; no temporal operators are employed in these works. In our opinion, temporal logic provides key benefits when verifying concurrent and reactive programs, in particular with respect to automatic verification, where the model checking community has demonstrated relative success when verifying hardware and embedded systems [Baier and Katoen, 2008; Clarke et al., 2001]. Also, it is important to note that, in the cited works, the authors assume a linear view of time when specifying systems and properties. In our approach, we focus on branching time properties of programs. In our opinion, branching time is important for fault-tolerance specification. This view is also shared by Attie, Arora, and Emerson in [Attie et al., 2004], where an algorithm for synthesizing fault-tolerant programs from CTL specifications is presented. They consider CTL as the temporal logic specification for the input of their synthesis method. Instead of using CTL, we use a branching time temporal logic that has a convenient mechanism for stating fault-tolerance properties, via the use of deontic operators. We believe that our formalism is better suited for capturing fault-tolerance properties, since the distinction between good, normative, ideal behavior and bad, faulty or unexpected executions is made naturally by using the deontic operators.

Additionally, we mention the work presented in [Janowski, 1995, 1997], where various notions of bisimulation are investigated with the aim of capturing fault-tolerant properties, in the context of process algebras. An obvious difference with respect to our work is that we use a state based approach and a temporal logic to reason about state based models, in contrast to the aforementioned works where process

algebras are employed for modeling systems, and the associated logic is a variation of Hennesy-Milner logic, which is known to be less expressive than temporal logics. Also, the notions of masking, nonmasking and failsafe fault-tolerance are not investigated in the referenced works.

Related to the second main contribution of our work, the synthesis method, various approaches have been proposed for synthesis of reactive systems from temporal logic specifications. The initial work was presented by Emerson and Clarke [Clarke and Emerson, 1981]. Their synthesis method was based on a decision procedure for checking the satisfiability of a CTL temporal logic specification. With respect to automated synthesis of fault-tolerant systems, the seminal work in this area is due to Attie, Arora, and Emerson [Attie et al., 2004], where they presented an algorithm for synthesizing fault-tolerant programs from CTL specifications, based on a tableau method defined by Emerson and Clarke in [Clarke and Emerson, 1981]. One main difference with our work is that we use deontic operators to distinguish between good and bad systems behavior, while in [Attie et al., 2004] the abnormal behavior is captured by means of faulty actions. Another difference with our work is that in [Attie et al., 2004] safety properties only need to hold after faults or through fail-free paths, which implies that the semantics of CTL has to be adapted to cope with this condition. Another important stream of work is presented in [Bonakdarpour et al., 2012]. Therein, Unity style programs are developed, the Unity logic being used to specify programs and to state fault-tolerant properties. Moreover, only a finite number of faults are allowed. It is important to notice that a main difference between that work and our approach is that our synthesized programs preserve *all* safety and liveness properties of the non-faulty part of the obtained program, while both [Bonakdarpour et al., 2012] and [Attie et al., 2004] preserve only the properties explicitly stated in the specification.

Another important issue in our research was how faults are injected during the synthesis process in order to produce a program which tolerates those faults, satisfying one of the levels of fault-tolerance. In our investigation, we observed that in many approaches (e.g., [Attie et al., 2004; Kulkarni and Arora, 2000; Kulkarni and Ebnenasir, 2004; Ebnenasir et al., 2008; Bonakdarpour et al., 2012]) faults are given

explicitly as part of the behavior model of the system. In more detail, they require as input a fault specification in their synthesis method, that is, a list of fault actions in a guarded command style. In our approach, we also require this in a similar way, but using CTL formulas. However, this is an optional input for the users in our synthesis algorithm. Additionally, we have proposed an automatic technique to inject faults through the negation of deontic propositional variables. We have obtained interesting results specially for masking and failsafe fault-tolerance when redundancy, consensus, agreement, and majority voting techniques are involved.

Finally, another difference between Attie et al. [Attie et al., 2004] and our work is how the required user-level of fault-tolerance (e.g., masking, nonmasking, failsafe) is checked during the synthesis process. In [Attie et al., 2004], they test it through CTL formulas if the level of fault-tolerance holds in the perturbed states as part of the tableau construction using the CTL decision procedure. However, in our case, we do not add any additional test during the tableau construction, we only use the dCTL- decision procedure for checking the satisfiability of the normal part of the given specification. Additionally, after the injection of faults, we perform the analysis of fault using of our simulation algorithms for pruning those states that do not satisfy the required level of fault-tolerance.

6.2 Contributions

We have presented a characterization of different levels of fault-tolerance by means of simulation relations. This formalization is simple and uses standard notions of simulation relations, by relating an operational system specification and a corresponding fault-tolerant implementation. Moreover, our approach to capturing fault-tolerance enables us to automatically verify, for example, that a given implementation of a system masks certain faults, or recovers from these faults, by employing variants of traditional bisimulation algorithms in our context. Indeed, we have adapted well known (bi)simulation algorithms to our setting, so that one can automatically check if a system implementation exhibits some degree of fault-tolerance. We have also studied the complexity of the resulting algorithms, and proved that they preserve the time complexity of traditional bisimulation algorithms. We have also studied properties of

our formalizations of fault-tolerance, showing that different kinds of temporal properties are preserved, depending on the degree of fault-tolerance that a system exhibits. Moreover, we have also presented results relating the different kinds of fault-tolerance.

We have proposed an approach to synthesizing fault-tolerant components from dCTL- specifications. dCTL- is a branching time temporal logic equipped with deontic operators, which is specially designed for fault-tolerant component specification. We believe this logic is better suited for fault-tolerance specification, and therefore synthesizing fault-tolerant implementations from dCTL- specifications is relevant. In order to capture fault-tolerance, we use an approach based on defining appropriate (bi)simulation relations, describing the relationship that must hold between a system specification and its fault-tolerant implementation. Our mechanism for synthesis is then based on combining decision procedures for the satisfiability of dCTL- temporal formulas, with (bi)simulation algorithms for checking a user required level of fault-tolerance.

Finally, we have implemented a tool called syntdctl for synthesizing masking and failsafe fault-tolerant problems. We have validated our synthesis method through several experiments such as, the Byzantine agreement problem, N-Modular-Redundancy, a Simple Train System, etc.

6.3 Future Work

We propose the following future work as potential extensions of our current results:

Extend our framework to accommodates *multitolerance*. Arora and Kulkarni formalized this concept in [Arora and Kulkarni, 1998a]. Essentially, in multitolerance, the set of fault actions is divided into classes, and different fault classes may require diverse levels of fault-tolerance (masking, nonmasking, or failsafe). For example, one class of faults may require masking fault-tolerance, while another class may demand only failsafe fault-tolerance. Thereby, *"multitolerance refers to the ability of a system to tolerate multiple classes of faults, each in a possibly different way"*. More recently,

Kulkarni and Ebnenasir in [Kulkarni and Ebnenasir, 2004] addressed the problem of automated synthesis of multitolerant programs. In short, they proved that if one needs to add failsafe (nonmasking) fault-tolerance in respect of one class of faults and masking fault-tolerance regarding another class of faults, then such addition can be performed in polynomial-time in the size of the state space of the fault-intolerant program. Nevertheless, if one needs to add failsafe fault-tolerance regarding one class of faults and nonmasking fault-tolerance for another class of faults, then the problem results in an NP-complete algorithm.

Extend our approach to be able to extract several concurrent components from a deontic logic specification. In this dissertation, we have been concerned with the synthesis of a single component. One possible way to incorporate concurrent components in our framework is by using indexes as is done in [Attie et al., 2004]. In this context, an issue to take into account is the preservation of the locality of the faults, i.e., if some faults affect one component, then the malfunction of this component do not add more faults to the states of other components.

An interesting problem to study is the property of *self-stabilizing* systems, i.e., those systems that eventually reach a global configuration from where all its behaviors are legitimate [Dolev, 2000]. For example, such a property is highly desirable in network protocols. Many researchers have pointed out the difficulties of designing and verifying self-stabilization systems since Dijkstra introduced it in the early seventies [Dijkstra, 1974]. Most of the existing techniques are either brute force, or adopt manual approaches non-amenable to automation. Recently, Farahat in [Farahat, 2012] developed and implemented a framework of heuristics for adding stabilization to distributed algorithms like token-ring, matching, leader election and consensus.

Another direction is to explore the extension of our work on synthesis of fault-tolerant programs to a probabilistic setting. We should start extending dCTL- allowing probabilistic quantification of described properties, similar to PCTL (Probabilistic Computation Tree Logic, see [Baier and Katoen, 2008]). Moreover, appropriate bisimulation relations for masking, nonmasking, and failsafe fault-tolerance also need to be extended, where each kind of fault occurs with a certain probability. A good basis

for extending our work to this setting is the work of Pantelic et al. in [Vera Pantelic, 2014].

Finally, in this dissertation, we have presented experimental results using our syntdctl tool, in particular, for the cases of masking and failsafe fault-tolerance. We left the implementation of nonmasking fault-tolerance for further work. Additionally, we want to provide a more concrete output format for our synthesis method. Currently, for a given dCTL- specification, we produce as output a state transition system where normal and faulty states are distinguished. Then, the idea is to synthesize a fault-tolerant program written in an output format such as modern language in which normal, faulty, and recovery actions can be expressed in some way.

Bibliography

Abadi, M., Lamport, L., and Wolper, P. (1989). Realizable and unrealizable specifications of reactive systems. In Ausiello et al. [1989], pages 1–17.

Abdelouahab, Z. and Braga, R. I. (2008). An adaptive train traffic controller. In *An Adaptive Train Traffic Controller, Springer Netherlands*, pages 550–555.

Abrial, J.-R. (2006). Train systems. In Butler, M. J., Jones, C. B., Romanovsky, A., and Troubitsyna, E., editors, *Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project], RODIN Book*, volume 4157 of *Lecture Notes in Computer Science*, pages 1–36. Springer.

Abrial, J.-R. and Hallerstede, S. (2007). Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundam. Inform.*, 77(1-2):1–28.

Abujarad, F. and Kulkarni, S. S. (2008). Automated addition of fault-tolerance to SCR toolset: A case study. In *28th IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2008 Workshops)*, pages 539–544. IEEE Computer Society.

Alpern, B. and Schneider, F. B. (1985). Defining liveness. *Inf. Process. Lett.*, 21(4):181–185.

Alur, R., Feder, T., and Henzinger, T. A. (1996). The benefits of relaxing punctuality. *J. ACM*, 43(1):116–146.

Alur, R. and La Torre, S. (2004). Deterministic generators and games for LTL fragments. *ACM Trans. Comput. Log.*, 5(1):1–25.

Amadio, R. M. and Prasad, S. (1994). Localities and failures (extended abstract). In Thiagarajan, P. S., editor, *Foundations of Software Technology and Theoretical Computer Science, 14th Conference, FSTTCS 1994*, volume 880 of *Lecture Notes in Computer Science*, pages 205–216. Springer.

Aminof, B., Ball, T., and Kupferman, O. (2004). Reasoning about systems with transition fairness. In Baader, F. and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004*, volume 3452 of *Lecture Notes in Computer Science*, pages 194–208. Springer.

Arora, A. (1992). *A Foundation of Fault-Tolerant Computing*. PhD thesis, The University of Texas at Austin.

Arora, A. and Gouda, M. G. (1993). Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027.

Arora, A. and Kulkarni, S. S. (1998a). Component based design of multitolerant systems. *IEEE Trans. Software Eng.*, 24(1):63–78.

Arora, A. and Kulkarni, S. S. (1998b). Detectors and Correctors: A theory of fault-tolerance components. In *18th International Conference on Distributed Computing Systems, ICDCS 1998*, pages 436–443. IEEE Computer Society.

Attie, P. C. (1999). Synthesis of large concurrent programs via pairwise composition. In Baeten, J. C. M. and Mauw, S., editors, *Concurrency Theory, 10th International Conference, CONCUR 1999*, volume 1664 of *Lecture Notes in Computer Science*, pages 130–145. Springer.

Attie, P. C., Arora, A., and Emerson, E. A. (2004). Synthesis of fault-tolerant concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(1):125–185.

Attie, P. C. and Emerson, E. A. (1998). Synthesis of concurrent systems with many similar processes. *ACM Trans. Program. Lang. Syst.*, 20(1):51–115.

Attie, P. C. and Emerson, E. A. (2001). Synthesis of concurrent programs for an atomic read/write model of computation. *ACM Trans. Program. Lang. Syst.*, 23(2):187–242.

Ausiello, G., Dezani-Ciancaglini, M., and Rocca, S. R. D., editors (1989). *Automata, Languages and Programming, 16th International Colloquium, ICALP 1989, Stresa, Italy, July 11-15, 1989, Proceedings*, volume 372 of *Lecture Notes in Computer Science*. Springer.

Avizienis, A., Laprie, J.-C., and Randell, B. (2004a). Dependability and its threats - a taxonomy. In Jacquart, R., editor, *IFIP Congress Topical Sessions*, pages 91–120. Kluwer.

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. E. (2004b). Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33.

Avizienis, A. A. (1995). *Software Fault Tolerance*, volume 2, chapter The Methodology of N-Version Programming, pages 22–45. John Wiley & Sons.

Baier, C. and Katoen, J.-P. (2008). *Principles of model checking*. MIT Press.

Barsotti, D., Nieto, L. P., and Tiu, A. (2007). Verification of clock synchronization algorithms: experiments on a combination of deductive tools. *Formal Asp. Comput.*, 19(3):321–341.

Bernardeschi, C., Fantechi, A., and Gnesi, S. (2002). Model checking fault tolerant systems. *Softw. Test., Verif. Reliab.*, 12(4):251–275.

Bernardeschi, C., Fantechi, A., Gnesi, S., and Santone, A. (1998). Formal validation of fault-tolerance mechanisms. In *Fast Abstract, The 28th International Symposium on Fault-Tolerant Computing, FTCS 1998*, pages 66–67.

Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., and Sa'ar, Y. (2012). Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938.

Bonakdarpour, B. (2008). *Automated Revision of Distributed and Real-Time Programs*. PhD thesis, Michigan State University.

Bonakdarpour, B., Kulkarni, S. S., and Abujarad, F. (2012). Symbolic synthesis of masking fault-tolerant distributed programs. *Distributed Computing*, 25(1):83–108.

Browne, M. C., Clarke, E. M., and Grumberg, O. (1987). Characterizing Kripke structures in temporal logic. In Ehrig, H., Kowalski, R. A., Levi, G., and Montanari, U., editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development, TAPSOFT 1987, Vol.1*, volume 249 of *Lecture Notes in Computer Science*, pages 256–270. Springer.

Bruns, G. (1992). A case study in safety-critical design. In von Bochmann, G. and Probst, D. K., editors, *Computer Aided Verification, Fourth International Workshop, CAV 1992*, volume 663 of *Lecture Notes in Computer Science*, pages 220–233. Springer.

Bruns, G. and Sutherland, I. (1997). Model checking and fault tolerance. In Johnson [1997], pages 45–59.

Büchi, J. R. and Landweber, L. H. (1969). Solving sequential conditions by finite state strategies. *Trans. Amer. Math. Soc.*, 138(2):295–311.

Carmo, J. and Jones, A. J. I. (1996a). Deontic database constraints, violation and recovery. *Studia Logica*, 57(1):139–165.

Carmo, J. and Jones, A. J. I. (1996b). Deontic database constraints, violation and recovery. *Studia Logica*, 57(1):139–165.

Castro, P. F. (2009). *Deontic Action Logics for Specification and Analysis of Fault-Tolerant Systems*. PhD thesis, McMaster University.

Castro, P. F., Kilmurray, C., Acosta, A., and Aguirre, N. (2011). dCTL: A branching time temporal logic for fault-tolerant system verification. In Barthe, G., Pardo, A., and Schneider, G., editors, *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011*, volume 7041 of *Lecture Notes in Computer Science*, pages 106–121. Springer.

Castro, P. F. and Maibaum, T. S. E. (2007). An ought-to-do deontic logic for reasoning about fault-tolerance: the diarrheic philosophers. In *Fifth IEEE International*

Conference on Software Engineering and Formal Methods, SEFM 2007, pages 151–160. IEEE Computer Society.

Castro, P. F. and Maibaum, T. S. E. (2009a). Deontic action logic, atomic boolean algebras and fault-tolerance. *J. Applied Logic*, 7(4):441–466.

Castro, P. F. and Maibaum, T. S. E. (2009b). Deontic logic, contrary to duty reasoning and fault tolerance. *Electr. Notes Theor. Comput. Sci.*, 258(2):17–34.

Castro, P. F. and Maibaum, T. S. E. (2009c). Reasoning about system-degradation and fault-recovery with deontic logic. In *Methods, Models and Tools for Fault Tolerance*, pages 25–43.

Castro, P. F. and Maibaum, T. S. E. (2010). Towards a first-order deontic action logic. In Mossakowski, T. and Kreowski, H.-J., editors, *Recent Trends in Algebraic Development Techniques - 20th International Workshop, WADT 2010*, volume 7137 of *Lecture Notes in Computer Science*, pages 61–75. Springer.

Chandy, K. M. and Misra, J. (1989). *Parallel program design - a foundation*. Addison-Wesley.

Chaochen, Z. and Hansen, M. R. (2004). *Duration Calculus - A Formal Approach to Real-Time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer.

Chaochen, Z., Hoare, C. A. R., and Ravn, A. P. (1991). A calculus of durations. *Inf. Process. Lett.*, 40(5):269–276.

Cho, K.-H. and Lim, J.-T. (1998). Synthesis of fault-tolerant supervisor for automated manufacturing systems: a case study on photolithographic process. *IEEE T. Robotics and Automation*, 14(2):348–351.

Chou, T. C. K. (1997). Beyond fault tolerance. *IEEE Computer*, 30(4):47–49.

Church, A. (1963). Logic, arithmetic, and automata. In *Proc. Internat. Congr. Mathematicians (Stockholm, 1962)*, pages 23–35. Inst. Mittag-Leffler.

Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M. (2000). NUSMV: A new symbolic model checker. *STTT*, 2(4):410–425.

Clarke, E. M. and Emerson, E. A. (1981). Design and synthesis of synchronization skeletons using branching-time temporal logic. In Kozen, D., editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer.

Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263.

Clarke, E. M., Grumberg, O., and Peled, D. (2001). *Model checking*. MIT Press.

Cristian, F. (1985). A rigorous approach to fault-tolerant programming. *IEEE Trans. Software Eng.*, 11(1):23–31.

De Nicola, R. and Vaandrager, F. W. (1995). Three logics for branching bisimulation. *J. ACM*, 42(2):458–487.

Demasi, R. (2013). Synthesizing fault-tolerant programs from deontic logic specifications. In *28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013*, pages 750–753. IEEE.

Demasi, R., Castro, P. F., Maibaum, T. S. E., and Aguirre, N. (2013a). Characterizing fault-tolerant systems by means of simulation relations. In Johnsen, E. B. and Petre, L., editors, *Integrated Formal Methods, 10th International Conference, IFM 2013*, volume 7940 of *Lecture Notes in Computer Science*, pages 428–442. Springer.

Demasi, R., Castro, P. F., Maibaum, T. S. E., and Aguirre, N. (2013b). Synthesizing masking fault-tolerant systems from deontic specifications. In Hung, D. V. and Ogawa, M., editors, *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013*, volume 8172 of *Lecture Notes in Computer Science*, pages 163–177. Springer.

Demasi, R., Castro, P. F., Maibaum, T. S. E., and Aguirre, N. (2014). Simulation relations for fault-tolerance. In *submitted for journal publication*.

Denning, P. J. (1976). Fault tolerant operating systems. *ACM Comput. Surv.*, 8(4):359–389.

Dijkstra, E. W. (1972). *Structured Programming*. Academic Press Ltd., London, UK, UK.

Dijkstra, E. W. (1974). Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall.

D'Ippolito, N., Braberman, V. A., Piterman, N., and Uchitel, S. (2010). Synthesis of live behaviour models. In Roman, G.-C. and Sullivan, K. J., editors, *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT FSE 2010*, pages 77–86. ACM.

D'Ippolito, N., Braberman, V. A., Piterman, N., and Uchitel, S. (2011). Synthesis of live behaviour models for fallible domains. In Taylor, R. N., Gall, H., and Medvidovic, N., editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011*, pages 211–220. ACM.

Dolev, S. (2000). *Self-Stabilization*. MIT Press.

Ebnenasir, A. (2007). Diconic addition of failsafe fault-tolerance. In Stirewalt, R. E. K., Egyed, A., and Fischer, B., editors, *22nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2007*, pages 44–53. ACM.

Ebnenasir, A., Kulkarni, S. S., and Arora, A. (2008). FTSyn: a framework for automatic synthesis of fault-tolerance. *International Journal of Software Tools for Technology Transfer (STTT)*, 10(5):455–471.

Emerson, E. A. (1981). *Branching time temporal logic and the design of correct concurrent programs*. PhD thesis, Harvard University.

Emerson, E. A. and Clarke, E. M. (1980). Characterizing correctness properties of parallel programs using fixpoints. In de Bakker, J. W. and van Leeuwen, J., editors, *Automata, Languages and Programming, 7th Colloquium, ICALP 1980*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer.

Emerson, E. A. and Clarke, E. M. (1982). Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266.

Emerson, E. A. and Halpern, J. Y. (1986). "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178.

Ezekiel, J. and Lomuscio, A. (2009). Combining fault injection and model checking to verify fault tolerance in multi-agent systems. In Sierra, C., Castelfranchi, C., Decker, K. S., and Sichman, J. S., editors, *8th International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS 2009*, pages 113–120. IFAAMAS.

Farahat, A. M. (2012). *Automated Design of Self-Stabilization*. PhD thesis, Michigan Technological University.

Fiadeiro, J. L. and Maibaum, T. S. E. (1991). Temporal reasoning over deontic specifications. *J. Log. Comput.*, 1(3):357–395.

Gärtner, F. C. (1999a). Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31.

Gärtner, F. C. (1999b). Transformational approaches to the specification and verification of fault-tolerant systems: Formal background and classification. *J. UCS*, 5(10):668–692.

Gärtner, F. C. and Jhumka, A. (2004). Automating the addition of fail-safe fault-tolerance: Beyond fusion-closed specifications. In Lakhnech, Y. and Yovine, S., editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004*, volume 3253 of *Lecture Notes in Computer Science*, pages 183–198. Springer.

Gilmore, S., Hillston, J., Holton, R., and Rettelbach, M. (1995). Specifications in stochastic process algebra for a robot control problem. *International Journal of Production Research*, 34:1065–1080.

Girault, A. and Rutten, É. (2009). Automating the addition of fault tolerance with discrete controller synthesis. *Formal Methods in System Design*, 35(2):190–225.

Gnesi, S., Latella, D., Lenzini, G., Abbaneo, C., Amendola, A. M., and Marmo, P. (2000). An automatic SPIN validation of a safety critical railway control system. In *International Conference on Dependable Systems and Networks, DSN 2000*, pages 119–124. IEEE Computer Society.

Gnesi, S., Lenzini, G., and Martinelli, F. (2005). Logical specification and analysis of fault tolerant systems through partial model checking. *Electr. Notes Theor. Comput. Sci.*, 118:57–70.

Guelfi, N., Pelliccione, P., Muccini, H., and Romanovsky, A. (2007). *An Introduction to Software Engineering and Fault Tolerance*, chapter Software Engineering of Fault Tolerant Systems. Series on Software Engineering and Knowledge Eng.

Guiho, G. D. and Hennebert, C. (1990). SACEM software validation (experience report). In Valette, F.-R., Freeman, P. A., and Gaudel, M.-C., editors, *Proceedings of the 12th International Conference on Software Engineering, ICSE 1990*, pages 186–191. IEEE Computer Society.

Harding, A., Ryan, M., and Schobbens, P.-Y. (2005). A new algorithm for strategy synthesis in LTL games. In Halbwachs, N. and Zuck, L. D., editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005*, volume 3440 of *Lecture Notes in Computer Science*, pages 477–492. Springer.

He, J. and Hoare, C. A. R. (1987). Algebraic specification and proof of a distributed recovery algorithm. *Distributed Computing*, 2(1):1–12.

Henzinger, M. R., Henzinger, T. A., and Kopke, P. W. (1995). Computing simulations on finite and infinite graphs. In *36th Annual Symposium on Foundations of Computer Science, FOCS 1995*, pages 453–462. IEEE Computer Society.

Hickey, J. (1999). Fault-tolerant distributed theorem proving. In Ganzinger, H., editor, *16th International Conference on Automated Deduction, CADE 1999*, volume 1632 of *Lecture Notes in Computer Science*, pages 227–231. Springer.

Holzmann, G. J. (1997). The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295.

Jackson, D. (2006). *Software Abstractions - Logic, Language, and Analysis*. MIT Press.

Janowski, T. (1995). *Bisimulation and Fault-Tolerance*. PhD thesis, University of Warwick, United Kingdom.

Janowski, T. (1997). On bisimulation, fault-monotonicity and provable fault-tolerance. In Johnson [1997], pages 292–306.

Jeffords, R. D., Heitmeyer, C. L., Archer, M., and Leonard, E. I. (2009). A formal method for developing provably correct fault-tolerant systems using partial refinement and composition. In Cavalcanti, A. and Dams, D., editors, *Formal Methods, Second World Congress, FM 2009*, volume 5850 of *Lecture Notes in Computer Science*, pages 173–189. Springer.

John, A., Konnov, I., Schmid, U., Veith, H., and Widder, J. (2013a). Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *Formal Methods in Computer-Aided Design, FMCAD 2013*, pages 201–209. IEEE.

John, A., Konnov, I., Schmid, U., Veith, H., and Widder, J. (2013b). Towards modeling and model checking fault-tolerant distributed algorithms. In Bartocci, E. and Ramakrishnan, C. R., editors, *Model Checking Software - 20th International Symposium, SPIN 2013*, volume 7976 of *Lecture Notes in Computer Science*, pages 209–226. Springer.

Johnson, M., editor (1997). *Algebraic Methodology and Software Technology, 6th International Conference, AMAST 1997, Sydney, Australia, December 13-17, 1997, Proceedings*, volume 1349 of *Lecture Notes in Computer Science*. Springer.

Jr., J. K., Smith, B. T., and Wojcik, A. S. (1989). Formal verification of fault tolerance using theorem-proving techniques. *IEEE Trans. Computers*, 38(3):366–376.

Kang, E. and Jackson, D. (2008). Formal modeling and analysis of a flash filesystem in Alloy. In Börger, E., Butler, M. J., Bowen, J. P., and Boca, P., editors, *Abstract State Machines, B and Z, First International Conference, ABZ 2008*, volume 5238 of *Lecture Notes in Computer Science*, pages 294–308. Springer.

Katz, S. and Perry, K. J. (1993). Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26.

Kent, S., Maibaum, T. S. E., and Quirk, W. J. (1993). Formally specifying temporal constraints and error recovery. In *Proceedings of IEEE International Symposium on Requirements Engineering, RE 1993*, pages 208–215. IEEE.

Kent, S., Quirk, W. J., and Maibaum, T. S. (Forest Research Project, 1991). Specifying deontic behaviour in modal action logic. Technical report.

Khosla, S. (1989). *System Specification: A Deontic Approach.* PhD thesis, Imperial College.

Khosla, S. and Maibaum, T. S. E. (1987). The prescription and description of state based systems. In Banieqbal, B., Barringer, H., and Pnueli, A., editors, *Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 243–294. Springer.

Kripke, S. (1963). Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16:83–94.

Krishnan, P. (1994). A semantic characterisation for faults in replicated systems. *Theor. Comput. Sci.*, 128(1&2):159–177.

Kulkarni, S. S. and Arora, A. (2000). Automating the addition of fault-tolerance. In Joseph, M., editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium, FTRTFT 2000*, volume 1926 of *Lecture Notes in Computer Science*, pages 82–93. Springer.

Kulkarni, S. S., Arora, A., and Chippada, A. (2001). Polynomial time synthesis of byzantine agreement. In *20th Symposium on Reliable Distributed Systems, SRDS 2001*, pages 130–. IEEE Computer Society.

Kulkarni, S. S. and Ebnenasir, A. (2003). Enhancing the fault-tolerance of nonmasking programs. In *23rd International Conference on Distributed Computing Systems, ICDCS 2003*, pages 441–449. IEEE Computer Society.

Kulkarni, S. S. and Ebnenasir, A. (2004). Automated synthesis of multitolerance. In *2004 International Conference on Dependable Systems and Networks, DSN 2004*, pages 209–. IEEE Computer Society.

Kulkarni, S. S. and Ebnenasir, A. (2005). Complexity issues in automated synthesis of failsafe fault-tolerance. *IEEE Trans. Dependable Sec. Comput.*, 2(3):201–215.

Kulkarni, S. S., Rushby, J. M., and Shankar, N. (1999). A case-study in component-based mechanical verification of fault-tolerant programs. In Arora, A., editor, *1999 ICDCS Workshop on Self-stabilizing Systems, WSS 1999*, pages 33–40. IEEE Computer Society.

Kumar, R., Garg, V., and Marcus, S. I. (1991). On controllability and normality of discrete event dynamical systems. *Systems & Control Letters*, 17(3):157–168.

Kumar, R. and Shayman, M. A. (1997). Centralized and decentralized supervisory control of nondeterministic systems under partial observation. *SIAM Journal on Control and Optimization*, 35:363–383.

Kupferman, O., Madhusudan, P., Thiagarajan, P. S., and Vardi, M. Y. (2000). Open systems in reactive environments: Control and synthesis. In Palamidessi, C., editor, *Concurrency Theory, 11th International Conference, CONCUR 2000*, volume 1877 of *Lecture Notes in Computer Science*, pages 92–107. Springer.

Lafortune, S. and Lin, F. (1991). On tolerable and desirable behaviors in supervisory control of discrete event systems. *Discrete Event Dynamic Systems*, 1(1):61–92.

Lamport, L. (1980). "sometime" is sometimes "not never" - on the temporal logic of programs. In Abrahams, P. W., Lipton, R. J., and Bourne, S. R., editors, *ACM Symposium on Principles of Programming Languages, POPL 1980*, pages 174–185. ACM Press.

Lamport, L. (1985). Solved problems, unsolved problems and non-problems in concurrency. *Operating Systems Review*, 19(4):34–44.

Lamport, L. (1994). The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923.

Lamport, L. and Merz, S. (1994). Specifying and verifying fault-tolerant systems. In Langmaack, H., de Roever, W. P., and Vytopil, J., editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 1994*, volume 863 of *Lecture Notes in Computer Science*, pages 41–76. Springer.

Lamport, L., Shostak, R. E., and Pease, M. C. (1982). The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401.

Lee, P. A. and Anderson, T. (1990). *Fault Tolerance: Principles and Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition.

Lentfert, P. J. A. and Swierstra, S. D. (1993). Towards the formal design of self-stabilizing distributed algorithms. In Enjalbert, P., Finkel, A., and Wagner, K. W., editors, *10th Annual Symposium on Theoretical Aspects of Computer Science, STACS 1993*, volume 665 of *Lecture Notes in Computer Science*, pages 440–451. Springer.

Lin, F. and Wonham, W. M. (1990). Decentralized control and coordination of discrete-event systems with partial observation. *IEEE Transactions On Automatic Control*, 35(12):1330 – 1337.

Lincoln, P. and Rushby, J. M. (1993). The formal verification of an algorithm for interactive consistency under a hybrid fault model. In Courcoubetis, C., editor, *Computer Aided Verification, 5th International Conference, CAV 1993*, volume 697 of *Lecture Notes in Computer Science*, pages 292–304. Springer.

Liu, Z. and Joseph, M. (1992). Transformation of programs for fault-tolerance. *Formal Asp. Comput.*, 4(5):442–469.

Liu, Z. and Joseph, M. (1993). Specification and verification of recovery in asynchronous communicating systems. In Vytopil, J., editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 221, pages 137–165. Springer.

Lomuscio, A. and Raimondi, F. (2006). MCMAS: A model checker for multi-agent systems. In Hermanns, H. and Palsberg, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006*, volume 3920 of *Lecture Notes in Computer Science*, pages 450–454. Springer.

Lomuscio, A. and Sergot, M. J. (2004). A formalisation of violation, error recovery, and enforcement in the bit transmission problem. *J. Applied Logic*, 2(1):93–116.

Lyu, M. R., editor (1996). *Handbook of Software Reliability Engineering*. McGraw-Hill, Inc., Hightstown, NJ, USA.

Madhusudan, P. and Thiagarajan, P. S. (1998). Controllers for discrete event systems via morphisms. In Sangiorgi, D. and de Simone, R., editors, *Concurrency Theory, 9th International Conference, CONCUR 1998*, volume 1466 of *Lecture Notes in Computer Science*, pages 18–33. Springer.

Madhusudan, P. and Thiagarajan, P. S. (2001). Distributed controller synthesis for local specifications. In Orejas, F., Spirakis, P. G., and van Leeuwen, J., editors, *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001*, volume 2076 of *Lecture Notes in Computer Science*, pages 396–407. Springer.

Maibaum, T. S. E. and Turski, W. M. (1984). On what exactly is going on when software is developed step-by-step. In Straeter, T. A., Howden, W. E., and Rault, J.-C., editors, *Proceedings, 7th International Conference on Software Engineering, ICSE 1984*, pages 528–533. IEEE Computer Society.

Maler, O., Nickovic, D., and Pnueli, A. (2006). From MITL to timed automata. In Asarin, E. and Bouyer, P., editors, *Formal Modeling and Analysis of Timed Systems, 4th International Conference, FORMATS 2006*, volume 4202 of *Lecture Notes in Computer Science*, pages 274–289. Springer.

Maler, O., Pnueli, A., and Sifakis, J. (1995). On the synthesis of discrete controllers for timed systems (an extended abstract). In *Symposium on Theoretical Aspects of Computer Science, STACS 1995*, pages 229–242.

Manna, Z. and Pnueli, A. (1990). A hierarchy of temporal properties. In Dwork, C., editor, *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing, PODC 1989*, pages 377–410. ACM.

Manna, Z. and Pnueli, A. (1992). *The temporal logic of reactive and concurrent systems - specification*. Springer.

Manna, Z. and Wolper, P. (1984). Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68–93.

Mantel, H. and Gärtner, F. C. (2000). A case study in the mechanical verification of fault tolerance. *J. Exp. Theor. Artif. Intell.*, 12(4):473–487.

Marchand, H. and Samaan, M. (2000). Incremental design of a power transformer station controller using a controller synthesis methodology. *IEEE Trans. Software Eng.*, 26(8):729–741.

McMillan, K. L. (1992). The SMV system. Technical report.

Milner, R. (1980). *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer.

Moreau, L. (2006). A fault-tolerant directory service for mobile agents based on forwarding pointers. *Scalable Computing: Practice and Experience*, 7(4).

Owre, S., Rushby, J. M., and Shankar, N. (1992). PVS: A prototype verification system. In Kapur, D., editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY. Springer-Verlag.

Peled, D. and Joseph, M. (1994). A compositional framework for fault tolerance by specification transformation. *Theor. Comput. Sci.*, 128(1&2):99–125.

Peleska, J. (1991). Design and verification of fault tolerant systems with CSP. *Distributed Computing*, 5:95–106.

Peterson, I. (1996). *Fatal Defect: Chasing Killer Computer Bugs*. David Mckay.

Pnueli, A. (1977). The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, FOCS 1977*, pages 46–57. IEEE Computer Society.

Pnueli, A., Asarin, E., , Maler, O., and Sifakis, J. (1998). Controller synthesis for timed automata. *Proc. System Structure and Control. Elsevier*.

Pnueli, A. and Rosner, R. (1989a). On the synthesis of a reactive module. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages, POPL 1989*, pages 179–190. ACM Press.

Pnueli, A. and Rosner, R. (1989b). On the synthesis of an asynchronous reactive module. In Ausiello et al. [1989], pages 652–671.

Prasetya, I. S. W. B. and Swierstra, S. D. (2005). Formal design of self-stabilizing programs. *J. High Speed Networks*, 14(1):59–83.

Qadeer, S. and Shankar, N. (1998). Verifying a self-stabilizing mutual exclusion algorithm. In Gries, D. and de Roever, W. P., editors, *Programming Concepts and Methods, IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods, PROCOMET 1998*, volume 125 of *IFIP Conference Proceedings*, pages 424–443. Chapman & Hall.

Rabin, M. O. (1972). *Automata on infinite objects and Church's problem*. Regional conference series in mathematics. Providence, R.I. Published for the Conference Board of the Mathematical Sciences by the American Mathematical Society.

Ramadge, P. J. and Wonham, W. M. (1987). Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230.

Ramadge, P. J. and Wonham, W. M. (1989). The control of discrete event systems. *IEEE*, 77(1):81–98.

Randell, B. and Xu, J. (1994). The evolution of the recovery block concept. In *Software Fault Tolerance*, pages 1–22. John Wiley & Sons Ltd.

Riely, J. and Hennessy, M. (1997). Distributed processes and location failures (extended abstract). In Degano, P., Gorrieri, R., and Marchetti-Spaccamela, A., editors, *Automata, Languages and Programming, 24th International Colloquium, ICALP 1997*, volume 1256 of *Lecture Notes in Computer Science*, pages 471–481. Springer.

Rudie, K. and Wonham, W. M. (1992). Think globally, act locally: decentralized supervisory control. *IEEE Transactions On Automatic Control*, 37(11):1692 – 1708.

Rushby, J. (1999). Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660.

Schlichting, R. D. and Schneider, F. B. (1983). Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238.

Schneider, F., Easterbrook, S. M., Callahan, J. R., and Holzmann, G. J. (1998). Validating requirements for fault tolerant systems using model checking. In *3rd International Conference on Requirements Engineering, ICRE 1998*, pages 4–13. IEEE Computer Society.

Schneider, M. (1993). Self-stabilization. *ACM Comput. Surv.*, 25(1):45–67.

Siewiorek, D. P. and Swarz, R. S. (1998). *Reliable Computer Systems (3rd Ed.): Design and Evaluation*. A. K. Peters, Ltd., Natick, MA, USA.

Sinha, P. and Suri, N. (1999). On the use of formal techniques for analyzing dependable real-time protocols. In *Proceedings of the 20th IEEE Real-Time Systems Symposium, RTSS 1999*, pages 126–135. IEEE Computer Society.

Sistla, A. P. and Clarke, E. M. (1985). The complexity of propositional lineal temporal logics. *J. ACM*, 32(3):733–749.

Srivas, M. K. and Miller, S. P. (1996). Applying formal verification to the AAMP5 microprocessor: A case study in the industrial use of formal methods. *Formal Methods in System Design*, 8(2):153–188.

Thistle, J. G. and Lamouchi, H. M. (2009). Effective control synthesis for partially observed discrete-event systems. *SIAM J. Control and Optimization*, 48(3):1858–1887.

Thomas, W. (1990). Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 133–192.

Thomas, W. (1995). On the synthesis of strategies in infinite games. In *Symposium on Theoretical Aspects of Computer Science, STACS 1995*, pages 1–13.

Thomas, W. (2002). Infinite games and verification (extended abstract of a tutorial). In Brinksma, E. and Larsen, K. G., editors, *Computer Aided Verification, 14th International Conference, CAV 2002*, volume 2404 of *Lecture Notes in Computer Science*, pages 58–64. Springer.

Torres-Pomales, W. (2000). Software fault tolerance: A tutorial. Technical report, NASA Technical Memorandum TM-2000-210616.

van Benthem, J. (1999). Modality, bisimulation and interpolation in infinitary logic. *Ann. Pure Appl. Logic*, 96(1-3):29–41.

Vardi, M. Y. (2001). Branching vs. linear time: Final showdown. In Margaria, T. and Yi, W., editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer.

Vera Pantelic, Mark Lawford, S. P. (2014). A framework for supervisory control of probabilistic discrete event systems. In *12th IFAC - IEEE Workshop on Discrete Event Systems, WODES 2014*.

Wallmeier, N., Hütten, P., and Thomas, W. (2003). Symbolic synthesis of finite-state controllers for request-response specifications. In Ibarra, O. H. and Dang, Z., editors, *Implementation and Application of Automata, 8th International Conference, CIAA 2003*, volume 2759 of *Lecture Notes in Computer Science*, pages 11–22. Springer.

Wieringa, R. J. and Meyer, J.-J. (1993). Applications of deontic logic in computer science: A concise overview. In *Deontic Logic in Computer Science, Normative System Specification*.

Wong, K. C. and Wonham, W. M. (1998). Modular control and coordination of discrete-event systems. *Discrete Event Dynamic Systems*, 8(3):247–297.

Yadav, D. and Butler, M. (2009). Formal development of a total order broadcast for distributed transactions using Event-B. In Butler, M. J., Jones, C. B., Romanovsky, A., and Troubitsyna, E., editors, *Methods, Models and Tools for Fault Tolerance*, volume 5454 of *Lecture Notes in Computer Science*, pages 152–176. Springer.

Yokogawa, T., Tsuchiya, T., and Kikuno, T. (2001). Automatic verification of fault tolerance using model checking. In *8th Pacific Rim International Symposium on Dependable Computing, PRDC 2001*, pages 95–102. IEEE Computer Society.

Zhang, B. (2008). Formal analysis of a distributed fault tolerant clock synchronization algorithm for automotive communication systems. In *34th Euromicro Conference on Software Engineering and Advanced Applications, EUROMICRO-SEAA 2008*, pages 393–400. IEEE.