Automated Revision of Distributed and Real-Time Programs

By

Borzoo Bonakdarpour

A Dissertation

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

Department of Computer Science and Engineering

2008

Abstract

Automated Revision of Distributed and Real-Time Programs

By

Borzoo Bonakdarpour

This dissertation concentrates on the problem of automated revision of distributed and real-time programs that are correct-by-construction. In particular, our research addresses the following question: "if an existing program fails to satisfy a property, is it feasible to automatically revise the program inside its current state space and set of transitions, so that the revised program satisfies the failed property while it continues to satisfy its current properties?" The main focus of this dissertation is to identify instances where automated revision of distributed and real-time programs can be achieved efficiently (in polynomial-time) and where it is difficult (hard in some class of complexity). We study this problem in two broad contexts: (1) revision in *closed* systems where programs do not interact with the environment, and (2) revision in *open* systems where programs are subject to a set of uncontrollable *faults* imposed by the environment. We refer to the former problem as "addition of a property to the input program" and the latter as "addition of fault-tolerance to the input program".

We expect the concept of program revision to play a crucial role in the context of distributed and real-time programs, where non-determinism, race conditions, and time-predictability make it significantly difficult to assert program correctness. Since developing automated formal analysis methods that can handle any arbitrary property involves highly complex decision procedures, we focus on properties that are typically used in specifying distributed and real-time systems. For instance, we consider Unity properties and its variations, that are widely used in specifying requirements of distributed and hard real-time systems. We classify our results into three types: (1) polynomial-time sound and complete algorithms, (2) hardness results, and (3) sound efficient heuristics. We note that we also present some results in the context of untimed centralized programs due to the following two reasons: (1) such results provide a valuable insight into the impact of augmenting programs with the notion of time and distribution, and (2) a hardness result in the context of untimed centralized programs identifies a lower bound on the complexity of the corresponding problem in the context of real-time and distributed programs.

Throughout this dissertation, we focus on three types of programs: (1) untimed centralized, (2) untimed distributed, and (3) centralized real-time. The reason for omitting distributed real-time programs is due to the fact that the structure of such programs are very complex and, hence, their formal analysis involves highly complex decision procedures. Thus, it is more beneficial to study the effect of the notions of distribution and time on programs separately in order to identify the stumbling blocks.

Regarding addition of properties to programs in closed systems, we focus on UNITY *safety* and *progress* properties. Our interest in UNITY properties is due to the fact that they have been found highly expressive in specifying a large class of programs.

Regarding addition of fault-tolerance to existing fault-intolerant programs, we consider three levels of fault-tolerance, namely *failsafe*, *nonmasking*, and *masking*, based on satisfaction of safety and liveness properties in the presence of faults. In order to capture time-related behaviors of programs in the presence of faults, in this dissertation, we extend the levels of fault-tolerance. In particular, for failsafe and masking fault-tolerance, we consider two additional levels, namely *soft* and *hard*, based on satisfaction of timing constraints in the presence of faults. Moreover, for nonmasking and masking fault-tolerance, we require that recovery to the normal behavior of programs should be achieved in a bounded amount of time. In addition, we introduce the notion of bounded-time *phased recovery* and present synthesis methods for generating programs that require such recovery mechanism.

In order to deploy our theoretical results in practice, we address some of the implementation difficulties, such as the time complexity of decision procedures and also high space complexity known as the *state explosion problem*. To this end, we present symbolic (BDD-based) heuristics for revising programs in both closed and open systems with respect to safety and progress properties. Our experimental results on synthesis of a variety of distributed programs show a significant performance improvement by several orders of magnitude in terms of time and space. We also introduce distributed and parallel techniques to improve the performance of our revision methods even further. Finally, we introduce our tool SYCRAFT which is capable of adding fault-tolerance to moderate-sized fault-intolerant distributed programs. In summary, this dissertation concludes that automated revision of moderate-sized programs (reachable states of size 10^{50} and beyond) is feasible in both theory and practice.

To my beautiful wife, Maryam, and my parents Afagh and Hossein Bonakdarpour,
for their unconditional love, encouragement, and support.

Acknowledgments[1]

First of all I thank my advisor, Dr. Sandeep Kulkarni, for offering me technical, financial, and moral support during the four years of my Ph.D. program. He introduced me to the area of automate program synthesis and transformation. Much of the results reported in this thesis is inspired by my discussions with him about our ideas on developing a general framework for specifying and revising distributed and real-time programs. He helped me understand what research is and how to face and solve a new problem.

My dissertation guidance committee comprised of Dr. Sandeep Kulkarni, Dr. Betty Cheng, Dr. Laura Dillon, and Dr. Jonathan Hall. I am grateful that I had access to valuable guidance from all four of them. I express my thanks to Dr. Dillon and Dr. Cheng for their critical reading and interesting questions during my defense. I express my thanks to Dr. Hall for his valuable questions and comments on practical applications of my work.

Also, I would like to truly thank the Department of computer Science and Engineering at Michigan State University for offering me financial support through teaching assistantships for several semesters.

I am so grateful to Dr. Luca de Alfaro for hosting me as a visiting researcher at University of California at Santa Cruz in summer 2006. Luca is a leading researcher in the area of game theory and model checking and I learned a great deal from him during my visit.

It has been a great pleasure to work closely with Ali Ebnenasir and Fuad Abujarad. They co-authored multiple papers with me on revising UNITY programs and on distributed revision algorithms. It is impossible to mention their innumerable contributions towards my work.

My special thanks go to my colleagues at SENS laboratory Mahesh Arumugam, Sascha Konrad, and Amir Khakpour for their comments and proof reading of my papers. Finally, I would like to use this opportunity to thank all my friends on Michigan State University campus; because of them my stay here has been very enjoyable and memorable.

[1]This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF grant EIA-0130724, and a grant from Michigan State University.

Table of Contents

LIST OF TABLES xi

LIST OF FIGURES xii

I Background 1

1 Introduction 2
1.1 Challenges in Automated Program Synthesis 3
1.2 Thesis 5
1.3 Contributions 6
1.4 Outline of the Dissertation 10

2 Preliminary Concepts 12
2.1 Programs 12
2.1.1 Timed Guarded Commands 17
2.1.2 Example (Real-Time Traffic Controller) 17
2.1.3 Example (Distributed Byzantine Agreement) 19
2.2 Specifications 20
2.2.1 Example 20
2.3 Region Graphs 22

II Revising Programs in Closed Systems 24

3 The Revision Problem in Closed Systems 26
3.1 Basic Concepts 26
3.2 Problem Statement 29

4 Revising Untimed Centralized Programs 32
4.1 Adding a Single Progress and Multiple Safety Properties 32
4.1.1 Example: Readers-Writers Program 35
4.2 Adding Multiple Progress Properties 39
4.3 Adding a Single Leads-to Property with Maximum Non-determinism 42

5 Revising Distributed Programs 44
5.1 Adding UNITY Safety Properties to Distributed Programs 45
5.2 Adding UNITY Progress Properties to Distributed Programs 52
5.3 A Symbolic Heuristic for Adding Leads-To Properties 57

6 Revising Real-Time Programs 61
6.1 Adding a Single Bounded-Time Leads-to Property 62

6.1.1 Example: Real-Time Resource Allocation 66
6.2 Revising Real-Time UNITY Programs with Maximum Non-determinism 68
6.3 Adding Interval-Bounded Leads-to Properties 71

III Revising Programs in Open Systems 73

7 The Revision Problem in Open Systems 75
7.1 Basic Concepts 75
7.1.1 The Type of Specifications in Part III 77
7.1.2 Example 78
7.2 Fault Model and Fault-Tolerance 79
7.2.1 Fault Model 79
7.2.2 Levels of Fault-Tolerance 80
7.2.3 Example 83
7.3 Problem Statement 84

8 Synthesizing Real-Time Fault-Tolerant Programs 87
8.1 Case Study: Altitude Switch 88
8.2 Adding Nonmasking Fault-Tolerance 92
8.2.1 Adding Bounded-Time Recovery in the Presence of Faults 93
8.2.2 Adding Nonmasking Fault-Tolerance Using Bounded-Time Recovery 97
8.3 Adding Soft and Hard-Failsafe Fault-Tolerance 104
8.3.1 Adding Soft-Failsafe Fault-Tolerance 104
8.3.2 Adding Hard-Failsafe Fault-Tolerance with One Bounded Response Property 107
8.4 Adding Soft and Hard-Masking Fault-Tolerance 114
8.4.1 Adding Soft-Masking Fault-Tolerance 114
8.4.2 Adding Hard-Masking Fault-Tolerance 117

9 Synthesizing Bounded-Time Phased Recovery 122
9.1 Bounded-Time Phased Recovery 123
9.2 Complexity of Synthesizing Bounded-Time 2-Phase Recovery 124
9.3 A Sufficient Condition for a Polynomial-Time Solution 127
9.3.1 Example (cont'd) 132

10 Disassembling Real-Time Fault-Tolerant Programs 135
10.1 Basic Concepts and Assumptions 137
10.2 Detectors and Their Role in Hard-Masking Programs 139
10.2.1 Detectors 140
10.2.2 Containment of Detectors in Real-Time Programs 140
10.2.3 Example (cont'd) 143
10.2.4 The Necessity of Existence of Detectors in Hard-Masking Programs 144
10.3 δ-Correctors and Their Role in Hard-Masking Programs 148
10.3.1 Weak and Strong δ-Correctors 148
10.3.2 Containment of δ-Correctors in Real-Time Programs 149
10.3.3 Example (cont'd) 150
10.3.4 The Necessity of Existence of Strong δ-Correctors in Hard-Masking Programs 151
10.3.5 The Necessity of Existence of Weak δ-Correctors in Hard-Masking Programs 156

11 Symbolic Synthesis of Distributed Fault-Tolerant Programs **158**
11.1 The Symbolic Synthesis Algorithm 160
11.2 Case Study 1: Byzantine Agreement 169
11.3 Case Study 2: Exploiting Human Knowledge to Assist Synthesis Algorithms . . 175
11.4 Case Study 3: Byzantine Agreement with Fail-Stop Faults 177
11.5 Case Study 4: Token Ring 180
11.5.1 The Effect of Multi-Step Recovery 184
11.6 Case Study 5: Infuse 185

12 The Tool SYCRAFT **191**
12.1 SYCRAFT Input Program Language and Grammar 192
12.1.1 Program, Constant, and Variable Declaration 193
12.1.2 Process Declaration and Structure 195
12.1.3 Invariant, Safety Specification, and Prohibited Transitions Declaration 202
12.1.4 Operator Precedence 203
12.2 Internal Functionality 203
12.3 Output Format 205
12.4 Example 1: Never 7 205
12.5 Example 2: Token Ring 208
12.6 Example 3: Byzantine Agreement 210

IV Distributed and Parallel Revision Techniques **215**

13 Distributed Synthesis of Centralized Fault-Tolerant Programs **217**
13.1 Parallel Construction of State Space 219
13.2 Distributed Addition of Failsafe Fault-Tolerance 220
13.3 Distributed Addition of Masking Fault-Tolerance 227

14 Parallelizing Symbolic Deadlock Resolution **233**
14.1 The Deadlock Resolution Problem 233
14.2 Parallel Symbolic Resolution of Deadlock States 236
14.2.1 Parallel Addition of Safe Recovery 236
14.2.2 Parallel State Elimination 240

V Literature Survey and Conclusion **243**

15 Related Work **244**
15.1 Program Synthesis in Closed Systems 245
15.1.1 Comprehensive Synthesis 246
15.1.2 Program Repair and Correction 247
15.2 Program Synthesis in Open Systems 248
15.2.1 Automated Synthesis of Fault-Tolerance 248
15.2.2 Controller Synthesis 251
15.2.3 Game Theory 253
15.3 Synthesis Tools 254
15.4 Component-Based Analysis of Fault-Tolerant Programs 256

16 Conclusion and Future Work **259**
16.1 Contributions 260
16.2 Open Problems and Future Research Directions 262
16.2.1 Open Problems Related to Complexity of Synthesis 263
16.2.2 Open Problems on Improving the Performance of Existing Algorithms 264
16.2.3 Extending the Boundaries of SYCRAFT 267
16.3 Other Research Directions 269
16.3.1 Synthesizing Fault-Tolerant Hybrid Systems 269
16.3.2 Incorporating Machine Learning and Data Mining techniques 270
16.3.3 Revising Fault-Tolerant Distributed Systems in Epistemic Logic 270

APPENDICES **288**

A Summary of Notation **289**

List of Tables

5.1 Assignment of values to variables in proof of Theorem 5.1.1. 47
5.2 Assignment of values to variables in proof of Theorem 5.2.1. 53
5.3 Experimental results of the symbolic heuristic. 60

7.1 Levels of fault-tolerance. 81

List of Figures

5.1 Mapping SAT to addition of Unity safety properties. 49
5.2 The structure of the revised distributed program (safety property). 51
5.3 Mapping SAT to addition of a progress property. 54
5.4 The structure of the revised distributed program (progress property). 57

6.1 Region graph of the real-time resource allocation program. 67

8.1 Timed guarded commands of $\mathcal{AS}$. 89
8.2 Region graph of $\mathcal{AS}$. 91
8.3 Fault timed actions in $\mathcal{AS}$. 91
8.4 Region graph with respect to fault-span of $\mathcal{AS}$. 92
8.5 Adjusted shortest path. 94
8.6 Region graph of nonmasking $\mathcal{AS}$. 103
8.7 Recovery timed guarded commands of nonmasking $\mathcal{AS}$. 103
8.8 Region graph of hard-failsafe $\mathcal{AS}$. 113
8.9 Revised timed guarded commands of hard-failsafe $\mathcal{AS}$. 114
8.10 Region graph of soft-masking $\mathcal{AS}$. 116
8.11 Recovery and revised timed guarded commands of soft-masking $\mathcal{AS}$. 117
8.12 Mapping 3-SAT to addition of hard-masking fault-tolerance. 118
8.13 Partial structure of the hard-masking program. 120

9.1 Mapping 2-path problem to fault-tolerance synthesis. 126

11.1 Experimental results for the Byzantine agreement problem. 171
11.2 Experimental results for modified Byzantine agreement problem. 176
11.3 Experimental results for Byzantine agreement subject to fail-stop faults. . . 178
11.4 Experimental results for token ring mutual exclusion program. 182
11.5 Experimental results for token ring with single-step recovery. 185
11.6 Experimental results for the Infuse protocol in sensor networks. 187

12.1 A sample run snapshot of Sycraft. 204
12.2 Never7 program state-transition graph. 206
12.3 Never7 program as input to Sycraft. 207
12.4 Fault-tolerant Never7 state-transition graph. 208
12.5 Sycraft output: fault-tolerant Never7. 208
12.6 Token ring program as input to Sycraft. 209
12.7 Sycraft output: fault-tolerant token ring. 211
12.8 The Byzantine agreement problem as input to Sycraft. 212
12.9 Sycraft output: fault-tolerant Byzantine agreement. 214

14.1 Inconsistencies raised by concurrency. 241

Part I

Background

Chapter 1

Introduction

Asserting correctness in a program is the most important aspect and application of formal methods. Two approaches to achieve correctness automatically in system design are:

- correct-by-verification, and
- correct-by-construction.

Automated verification (and in particular model checking) is arguably one of the most successful contributions of formal methods in hardware and software development in the past three decades. However, if verification of a program against a mathematical model (e.g., a set of properties) identifies an error in the system, one needs to fix the error manually. Such manual revision inevitably requires another step of verification in order to ensure that the error is indeed resolved and that, no new errors are introduced to the program at hand. Thus, accomplishing correctness through verification involves a cycle of design, verification, and subsequently manual revision, if the verification step does not succeed.

Another common scenario is where requirements of a program evolve during the program life cycle. Evolution of requirements is largely due to two major factors: *change of environment* and the *incomplete specification* phenomenon. As a matter of fact, the latter has become a customary stumbling issue in virtually any design and development team. In this scenario, in order to maintain an existing program according to changes in specification, one needs to modify the program so that it satisfies additional properties of interest, while satisfying existing properties. Again, redesigning a program manually to address the above issues may be a tedious task, as it simply may not be successful and potentially introduces additional human errors to the existing program. This iterative procedure of verification and manual revision of programs often requires a vast amount of resources. In other words, achieving correctness by verification is an *after-the-fact* task, which may potentially be costly.

The above scenarios clearly motivate the need for automating the revision phase. An automated method should revise a program so that the output program preserves existing properties of the input program in addition to satisfying new properties. Using such revision, there is no need to re-verify the correctness of the revised program. In other words, the resulting program is correct-by-construction. Taking the paradigm of correct-by-construction to extreme leads us to automated *synthesis*, where a program is constructed from a set of properties from scratch. Alternatively, in *program revision*, an automated method transforms an input program into an output program that meets additional properties.

In the rest of this chapter, first in Section 1.1, we describe the challenges and demands in the broad context of program synthesis and in particular program revision. Then, we state the thesis that we defend in this dissertation in Section 1.2. We present the contributions of the dissertation in Section 1.3. Finally, we describe the outline of the dissertation in Section 1.4.

1.1 Challenges in Automated Program Synthesis

The seminal work on program verification and synthesis was published at about the same time [EC82, MW84, VW86, CES86]. However, while there have been impressive advances in theory of model checking and its deployment in practice, less attention has been paid to program synthesis in the formal methods community. We believe that there are two reasons for the existing gap between the state-of-the-art theory and practice of automated program synthesis as compared to verification:

1. In verification, we often verify one property at a time, whereas in synthesis, we start with a *set* of temporal assertions. More precisely, in verification, complex structure of one property does not affect verification of other easy-to-handle properties. However, in synthesis, this is obviously not the case [KPV06].

2. The complexity of decision procedures in synthesis algorithms are generally higher than those in verification algorithms. One may argue that this reason is not com-

pelling, as the complexity of synthesis algorithms are measured in the size of formulae to be synthesized and the complexity of verification algorithms are measured in the size of the given model. In addition, one may argue that the worst case complexity in both verification and synthesis is rarely raised. However, we note that there exist cases, where synthesizing a property is undecidable, whereas its verification is decidable [AH93]. In such cases, the complexity of decision procedure of a synthesis algorithm has a significant impact on feasibility of synthesis.

Two broad approaches for dealing with automated synthesis are the following:

1. In *comprehensive synthesis*, the designer develops a new program from scratch that implements a set of properties. Several approaches exist in the context of comprehensive synthesis [EC82, MW84, AE01, LW90, LL92, RLL03, Roh04, WHT03, Tho02, MNP06, AFH96] most of which require a complete specification and are based on developing a satisfiability proof of a formula. Comprehensive synthesis also allows less emphasis on *reuse* in the face of constantly changing program specifications.

2. In *program revision* (also called *local redesign* or *program repair*) [BEKar, BK06a, JGB05, EKB05], the designer *revises* an existing program by removing the behaviors that violate a property of interest without adding any new behaviors. In other words, program revision is an automated method that transforms an input program into an output program that meets additional properties.

Comprehensive synthesis has mostly been studied in two contexts: (1) synthesis from specification, and (2) synthesis via solving games. In the former, the synthesis problem reduces to the satisfiability problem of the corresponding specification language. Such methods only address synthesis of *closed* systems, where the system does not interact with the environment. In the latter, the synthesis problem reduces to the realizability problem [ALW89] of the corresponding specification language. The realizability problem considers *open* systems, where the system interacts with the environment. In reactive systems, the synthesis problem is viewed as identifying a winning strategy in a game between the system and the environment. Such methods transform the specification into a parity automaton [EJ91] over unfolded programs represented as infinite trees and checks whether such an automaton is nonempty, i.e., it accepts some infinite tree [PR89a].

One drawback of specification-based synthesis methods, where we start synthesizing with a set of properties, is that any change in the specification requires us to redo synthesis from scratch. The difficulty with tree automata-theoretic methods is that they require determinization of Büchi automata. Such determinization in turn exhibits the following two obstacles: (1) it has been shown to be resistant to efficient implementation [ATW05, THB95], and (2) it results with a very complicated state space [Jur00], which is not suited for optimized implementation such as symbolic techniques.

While synthesis from specification is undoubtedly useful in cases where no program exists to start from, it suffers from lack of *reuse* and resistance to efficient implementation. Alternatively, program revision may remedy both shortcomings of comprehensive synthesis. In this dissertation, we focus on automated revision of distributed and real-time programs.

1.2 Thesis

Comprehensive approaches mentioned in Section 1.1 play an important role in the broad area of program synthesis. However, in the face of increasingly dynamic systems with evolving requirements, program revision is highly desirable due to the following reasons:

1. It has the potential to *reuse* certain computations of an existing program without actually exacerbating the complexity of the state space explosion problem (unlike automata-theoretic approaches).

2. It can be applied in cases where a complete specification of a program is not available.

3. It can also be applied in cases where the existing program satisfies properties whose automated synthesis is undecidable or lies in highly complex classes of complexity. One example of a property whose synthesis is undecidable is the so-called real-time *punctuality property* [AFH96]; a property with precise eventuality $\Diamond_{=\delta} Q$.

4. It retains the current structure of the program to be revised and *add* the newly identified requirements to the program in an incremental fashion. Such incremental synthesis is especially useful when the original program is designed manually, e.g., for ensuring better efficiency.

Although program revision presents potential useful applications, the question of its feasibility remains open in the literature. With this motivation, we focus on complexity analysis and consequently practicality of program revision. In fact, in this dissertation, we propose and defend the following thesis:

Automated revision of moderate-sized distributed and real-time programs is feasible.

The contributions made in this dissertation validate this thesis. Briefly, based on the structure of the input program and the property of interest, we classify our contributions into cases where (1) polynomial-time sound and complete revision algorithm exists, (2) the time complexity of revision is exponential, and (3) efficient revision via developing heuristics is possible at the cost of losing completeness.

1.3 Contributions

In this dissertation, we study the problem of automated revision of *distributed* and *real-time* programs. We require that such revision must result in a program that continues to satisfy all universally quantified properties of the original program. Since developing formal analysis methods that can handle any arbitrary property requires dealing with highly complex decision procedures, we focus on properties and requirements that are typically used in specifying real-time and distributed systems. For instance, we consider untimed and real-time UNITY [CM88] properties and their variations. As mentioned earlier, in order to address the feasibility of automated program revision, we classify our results into the following three types:

- **Polynomial-time sound and complete revision algorithms.** A revision algorithm would be especially useful, if it were sound and complete. A *sound* algorithm ensures that the revised program meets the new specification (in addition to preserving all properties of the original program). This requirement essentially implies that the revised program is correct-by-construction. Moreover, a *complete* algorithm provides designers with insight to decide whether a program can be revised as it is, or if it needs to be redesigned from scratch. This is because when a complete revision algorithm concludes that a given program cannot be revised, it means that all behaviors of the input program contradict the new specification. Such automated assistance is highly desirable for a designer, as it significantly decreases the design time by warning the designer about spending time on fixing a program that is not *fixable* [BK06a, EKB05].

- **Identifying complexity hierarchy.** The knowledge of complexity bounds is especially important in building tools for automated program revision. For instance, an NP-completeness result demonstrates that corresponding tools must utilize efficient heuristics to expedite the revision algorithm at the cost of completeness. Moreover, hardness proofs often identify where the exponential complexity lies in the problem. Thus, thorough analysis of proofs is also crucial in devising efficient heuristics.

- **Efficient revision heuristics.** The complexity of automated synthesis can be characterized in two parts. The first part has to deal with questions such as *which* transitions/states should be added to the input program, and *which* transitions/states should be removed from the input program. The second part has to deal with questions such as *how quickly* such addition and removal can be achieved. In particular, when a revision problem involves exponential complexity, we need to devise heuristics that can solve the problem in polynomial-time at the cost of completeness. Such heuristics are often the core of tools for automated formal analysis.

We also present some results in the context of untimed centralized programs for two reasons:

1. such results provide a valuable insight into the impact of augmenting programs with the notion of time and distribution, and

2. a hardness result for revising untimed centralized programs with respect to a particular property identifies a lower bound on the complexity of the corresponding problem in the context of distributed and real-time programs.

Our revision approach is graph-theoretic in the sense that we first transform a program into a directed state-transition graph. In case of real-time programs, the graph is weighted. Then, using the standard algorithms in graph theory, we manipulate the graph such that the program associated with the resultant graph meets the desired properties.

We study the revision problem from four different perspectives described next.

1. *Adding properties to existing distributed, untimed, and real-time programs*

This part of our research addresses automated revision of programs in closed systems [BEKar, BK08b, BK06a, EKB05]. In other words, we consider cases where the given program does not interact with the environment. In particular, we study the complexity of adding different types of UNITY properties to existing untimed centralized, distributed, and real-time programs. UNITY properties are highly expressive for specifying distributed and real-time programs [CM88]. We show that the complexity of adding UNITY properties significantly varies in the context of untimed centralized, distributed, and real-time programs. We also show that small changes in constraints of a property (e.g., specifying a lower bound in a bounded-time leads-to property) may considerably change the complexity hierarchy of revision. Moreover, we show that imposing small constraints on the semantics of the revised program (e.g., adding a property while preserving maximum non-determinism) can also significantly change the complexity hierarchy of revision.

2. *Adding fault-tolerance to existing fault-intolerant real-time programs*

   This part of our research addresses revision of programs in reactive systems where the program interacts with the environment [BK06b, BKA08, BK08a]. We model such interactions by a set of *faults*, which cause either unexpected state perturbations or time delays. In order to characterize fault-tolerance requirements of programs, we consider three levels of fault-tolerance. These levels are *nonmasking* (respectively, stabilizing), *failsafe*, and *masking*, based on satisfaction of safety and liveness properties in the presence of faults [AG93, Kul99]. Furthermore, in order to capture the time-related behaviors of programs in the presence of faults, we propose two additional levels, namely, *soft* and *hard* fault-tolerance, based on satisfaction of timing constraints in the presence of faults. Intuitively, in the absence of faults, both soft and hard fault-tolerant programs are required to satisfy their timing constraints. However, in the presence of faults, a soft fault-tolerant program is *not* required to satisfy its timing constraints while a hard fault-tolerant program is required to do so. Moreover, for nonmasking and masking fault-tolerance, we require that recovery to the normal behavior of a program should be achieved within a bounded amount of time. Equipped with the notion of levels of fault-tolerance, we study the effect of restrictions of each level on the complexity of addition of fault-tolerance to existing fault-intolerant real-time programs.

3. *Developing techniques and tools for adding fault-tolerance to fault-intolerant programs in practice*

   As mentioned earlier, the complexity of automated synthesis of fault-tolerant programs can be characterized in two parts. The first part has to deal with questions such as *which* transitions/states should be added, and *which* transitions/states should be removed to prevent violation of specification in the presence of faults. The second part has to deal with questions such as *how quickly* such states and transitions can be identified. In the above two parts of our contributions, we focus on the first question to identify the complexity of various revision problems. Observe that the solution to the first question is independent of issues such as representation of programs, faults, specifications, etc. Hence, we utilize explicit-state (enumerative) techniques to develop algorithms. Explicit-state techniques are especially valuable in this context, as we can analyze how different heuristics affect a given program. This analysis enables us to identify bottlenecks and stumbling blocks of heuristics. Explicit-state techniques, however, are undesirable for the second part, as they suffer from the state explosion problem and prevent one from synthesizing programs where the state space is large. In order to develop efficient techniques, we also focus on the second part of the problem to improve the time and space complexity of synthesis. Towards this end, we focus on algorithms for adding fault-tolerance to *untimed distributed* fault-intolerant programs where programs, faults, specifications, etc., are modeled using Boolean formulae represented by Bryant's Ordered Binary Decision Diagrams [Bry86]. Our experimental results on a wide range of examples in fault-tolerant distributed computing shows significant performance improvement by several orders of magnitude over enumerative methods for moderate-sized state spaces (10^{50} and beyond) [BK07b].

   The tool SYCRAFT [BK08c] implements our BDD-based heuristics. In SYCRAFT, a distributed fault-intolerant program is specified in terms of a set of processes and an invariant. Each process is specified as its set of transitions, a set of variables that the process can read, and a set of variables that the process can write. Given a distributed fault-intolerant program, SYCRAFT generates a masking fault-tolerant distributed program.

4. *Developing distributed and parallel revision techniques*

   Time and space complexity has always been a major challenge in deployment of automated formal methods. Thus, exploiting multiple machines to expand available memory and multiple processors to increase computing power seems to be a sensible breakthrough. In fact, with recent advances in deployment of distributed systems for parallel processing and in particular multi-core processors, an increasing interest in parallel and distributed techniques has emerged in automated formal methods. Thus, as yet another means to improve the performance of automated revision, we explore the potential of using distributed and parallel techniques in automated program revision [BK07a, BAK08].

1.4 Outline of the Dissertation

This dissertation consists of five parts. Each part addresses a different aspect of automated program revision.

- Part I informally describes the revision problem and our contributions (Chapter 1) and then to formally presents the preliminary concepts (Chapter 2).

- Part II of the dissertation addresses revision of programs in *closed* systems. In this part, first, in Chapter 3, we formally define the revision problem in the context of closed systems. Then, in Chapters 4, 5, and 6, we present our contributions on automated revision of untimed centralized, distributed, and real-time programs in closed systems, respectively.

- In Part III, we focus on *open* systems. Specifically, Chapter 7 is dedicated to formally define the revision problem in the context of open systems. In Chapter 8, we present our results on automated synthesis of real-time fault-tolerant programs. We introduce the notion of bounded-time phased recovery and present complexity of it synthesis in Chapter 9. Chapter 10, presents our results on analyzing the structure and decomposition of fault-tolerant real-time programs. In Chapter 11, we shift our focus to distributed fault-tolerant programs. Specifically, we present our BBD-based heuristics for adding fault-tolerance to distributed programs. Finally, we present our tool SYCRAFT in Chapter 12.

- Part IV presents our distributed and parallel revision techniques. In Chapter 13, we introduce our distributed synthesis algorithms (distributed processor, distributed memory) for adding failsafe and masking fault-tolerance to untimed centralized programs. Then, in Chapter 14, we introduce our multi-core BDD-based algorithm for deadlock resolution.

- Finally, in Part V, we conclude. We present the related work in the literature of automated program synthesis in Chapter 15. Finally, in Chapter 16, we discuss open problems, 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 formal framework, namely, programs, specifications, and region graphs. Intuitively, in our framework, programs are specified in terms of their state space and their transitions. The notion of real-time programs is inspired by Alur and Henzinger [AH97]. Formalization of the issue of distribution is due to Kulkarni and Arora [KA00]. The definition of specification is adapted from Alpern and Schneider [AS85] and Henzinger [Hen92]. Finally, the notion of region graph is due to Alur and Dill [AD94]. Throughout this chapter, we incorporate two examples (real-time traffic controller and distributed Byzantine agreement) to illustrate the basic concepts.

2.1 Programs

In this section, we formally present the notion of *programs*. Intuitively, we define a program in terms of a set of processes. Each process is in turn specified by a *state-transition system* and is constrained by some *read/write restriction* over its set of discrete variables and *timing constraints* over its set of clock variables. Read/write restrictions are intended to model the issue of distribution and the timing constraints are meant to formalize real-time features of programs. Thus, our definition of programs covers the large class of distributed and real-time programs.

Let $V = \{v_0, v_2 \cdots v_n\}$, $n \geq 0$, be a finite set of *discrete variables* and $X = \{x_0, x_2 \cdots x_m\}$, $m \geq 0$, be a finite set of *clock variables*. Each discrete variable v_i, $0 \leq i \leq n$, is associated with a finite *domain* D_{v_i} of values. Each clock variable x_j, $0 \leq j \leq m$, ranges over nonnegative real numbers (denoted $\mathbb{R}_{\geq 0}$). A *location* is a function that maps the discrete variables in V to a value from their respective domain. A *clock constraint* over X is a Boolean combination of formulae of the form $x \preceq c$ or $x - y \preceq c$, where $x, y \in X$, $c \in \mathbb{Z}_{\geq 0}$, and $\preceq$ is either $<$ or $\leq$. We denote the set of all clock constraints over X by $\Phi(X)$.

A *clock valuation* is a function $\nu : X \rightarrow \mathbb{R}_{\geq 0}$ that assigns a real value to each clock variable. For $\tau \in \mathbb{R}_{\geq 0}$, we write $\nu + \tau$ to denote $\nu(x) + \tau$ for every clock variable x in X. Also, for $\lambda \subseteq X$, $\nu[\lambda := 0]$ denotes the clock valuation that assigns 0 to each $x \in \lambda$ and agrees with ν over the rest of the clock variables in X. We now define the notion of state, in which we specify the value of both discrete and clock variables.

Definition 2.1.1 (state) A *state* (denoted σ) is a pair (s, ν), where s is a location and ν is a clock valuation for X. ∎

Let u be a (discrete or clock) variable and σ be a state. We denote the value of u in state σ by $u(\sigma)$. The set of all possible states is called the *state space* (denoted $\mathcal{S}$) obtained from the associated variables and their corresponding domains.

Definition 2.1.2 (transition) A *transition* is an ordered pair (σ_0, σ_1), where σ_0 and σ_1 are two states. Transitions are classified into two types:

- *Immediate transitions:* $(s_0, \nu) \rightarrow (s_1, \nu[\lambda := 0])$, where s_0 and s_1 are two locations, ν is a clock valuation, and λ is a set of clock variables, where $\lambda \subseteq X$.

- *Delay transitions:* $(s, \nu) \rightarrow (s, \nu + \delta)$, where s is a location, ν is a clock valuation, and $\delta \in \mathbb{R}_{\geq 0}$ is a *time duration*. We denote a delay transition of duration δ at state σ by (σ, δ). ∎

Definition 2.1.3 (state predicate) Let $\mathcal{S}$ be the state space obtained from variables in V and X. A *state predicate* is a subset S of $\mathcal{S}$, such that if φ is a constraint involving clock variables in X, where $S \Rightarrow \varphi$, then $\varphi \in \Phi(X)$, i.e., in the corresponding Boolean expression of S, clock variables are only compared to nonnegative integers (and not real numbers). ∎

Definition 2.1.4 (transition predicate) Let $\mathcal{S}$ be the state space obtained from variables in V and X. A *transition predicate* is a subset of $\mathcal{S} \times \mathcal{S}$. Again, we require that in the corresponding Boolean expression that describes the set of source states, clock constraints are in $\Phi(X)$. ∎

Notation. Let T be a transition predicate. We let T^s and T^d denote the set of immediate and delay transitions in T, respectively, where $T = T^s \cup T^d$.

Definition 2.1.5 (process) A *process* p is specified by the tuple $\langle V_p, X_p, T_p, R_p, W_p \rangle$ where V_p is a set of discrete variables, X_p is a set of clock variables, T_p is a transition predicate in the state space of p (denoted $\mathcal{S}_p$), R_p is a set of variables that p can read, and W_p is a set of variables that p can write such that $W_p \subseteq R_p \subseteq V_p$ (i.e., we assume that p cannot blindly write a variable). ∎

We now formalize the issue of distribution in processes using restrictions on reading and writing discrete variables.

Write restrictions. Let $p = \langle V_p, X_p, T_p, R_p, W_p \rangle$ be a process. Clearly, T_p must be disjoint from the following transition predicate due to inability of p to change the value of variables that p cannot write:

$$NW_p = \{(\sigma_0, \sigma_1) \mid \exists v \in (V_p - W_p) \ : \ v(\sigma_0) \neq v(\sigma_1)\}.$$

Read restrictions. Let $p = \langle V_p, X_p, T_p, R_p, W_p \rangle$ be a process, v be a variable in V_p, and $(\sigma_0, \sigma_1) \in T_p$ where $\sigma_0 \neq \sigma_1$. If v is not in R_p, then T_p must include a corresponding transition from all states σ'_0 where σ'_0 and σ_0 differ only in the value of v. Let (σ'_0, σ'_1) be one such transition. Now, it must be the case that σ_1 and σ'_1 are identical except for the value of v, and, the value of v must be the same in σ'_0 and σ'_1. For instance, let $V_p = \{a, b\}$ and $R_p = \{a\}$. Since p cannot read b, the transition $(a = 0, b = 0) \rightarrow (a = 1, b = 0)$ and the transition $(a = 0, b = 1) \rightarrow (a = 1, b = 1)$ have the same effect as far as p is concerned. Thus, each transition (σ_0, σ_1) in T_p is associated with the following *group predicate*:

$$\begin{aligned} Group_p(\sigma_0, \sigma_1) = \{(\sigma'_0, \sigma'_1) \mid & \\ & (\forall v \in (V_p - R_p) \ : \ (v(\sigma_0) = v(\sigma_1) \ \wedge \ \ v(\sigma'_0) = v(\sigma'_1))) \ \wedge \\ & (\forall v \in R_p \qquad\quad : \ (v(\sigma_0) = v(\sigma'_0) \ \wedge \ \ v(\sigma_1) = v(\sigma'_1)))\}. \end{aligned}$$

Definition 2.1.6 (program) A *program* $\mathcal{P}$ is specified by the tuple $\langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$, where $\Pi_{\mathcal{P}}$ is a set of processes and $I_{\mathcal{P}}$ is a nonempty set of initial states. Without loss of generality, we assume that the state space of all processes in $\mathcal{P}$ are identical (i.e., $\forall p, q \in \Pi_{\mathcal{P}} :: (V_p = V_q) \wedge (X_p = X_q)$). Thus, the set of variables (denoted $V_{\mathcal{P}}$) and state space of program $\mathcal{P}$ (denoted $\mathcal{S}_{\mathcal{P}}$) is identical to the set of variables and state space of the processes of $\mathcal{P}$, respectively. In this sense, the set $I_{\mathcal{P}}$ of initial states of $\mathcal{P}$ is a subset of $\mathcal{S}_{\mathcal{P}}$. ∎

Notation. Let $\mathcal{P}$ be a program. The set $T_{\mathcal{P}}$ denotes the collection of transition predicates of all processes of $\mathcal{P}$, i.e., $T_{\mathcal{P}} = \bigcup_{p \in \mathcal{P}} T_p$.

Definition 2.1.7 (computation) A *computation* of a program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$ is a finite or infinite timed state sequence of the form:

$$\overline{\sigma} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$$

for which the following conditions are satisfied:

1. $\forall i \in \mathbb{Z}_{\geq 0} : \sigma_i \in \mathcal{S}_{\mathcal{P}}$,

2. $\forall j \in \mathbb{Z}_{\geq 0} : (\sigma_j, \sigma_{j+1}) \in T_{\mathcal{P}}$,

3. if $\overline{\sigma}$ reaches a terminating state σ_f where there does not exist σ such that $\sigma \neq \sigma_f$ and $(\sigma_f, \sigma) \in T^s_{\mathcal{P}}$ then we extend $\overline{\sigma}$ to an infinite computation by stuttering at σ_f and letting global time (i.e., the sequence $\tau_0\tau_1 \cdots$) advance indefinitely, and

4. the sequence $\tau_0\tau_1 \cdots$ (called the *global time*), where $\tau_i \in \mathbb{R}_{\geq 0}$ for all $i \in \mathbb{Z}_{\geq 0}$, satisfies the following constraints:

   - *(monotonicity)* for all $i \in \mathbb{Z}_{\geq 0}$, $\tau_i \leq \tau_{i+1}$,
   - *(divergence)* for all $t \in \mathbb{R}_{\geq 0}$, there exists $j \in \mathbb{Z}_{\geq 0}$ such that $\tau_j \geq t$, and
   - *(consistency)* for all $i \in \mathbb{Z}_{\geq 0}$, (1) if (σ_i, σ_{i+1}) is a delay transition, say (σ_i, δ), in $T^d_{\mathcal{P}}$, then $\tau_{i+1} - \tau_i = \delta$, and (2) if (σ_i, σ_{i+1}) is an immediate transition in $T^s_{\mathcal{P}}$, then $\tau_i = \tau_{i+1}$. ∎

We distinguish between a *terminating* computation and a *deadlocked* finite computation. Precisely, when a computation $\overline{\sigma}$ terminates in state σ_f, we include the delay transitions (σ_f, δ) in $T^d_{\mathcal{P}}$ for all $\delta \in \mathbb{R}_{\geq 0}$, i.e., $\overline{\sigma}$ can be extended to an infinite computation by advancing time arbitrarily. On the other hand, if there exists a state σ_d, such that there is no outgoing (delay or immediate) transition from σ_d, then σ_d is a *deadlock state.*

Notation. Let $\overline{\sigma}_i$ denote the pair (σ_i, τ_i) in computation $\overline{\sigma}$. Also, let $\overline{\alpha}$ be a finite computation of length n and $\overline{\beta}$ be a finite or infinite computation. The *concatenation* of $\overline{\alpha}$ and $\overline{\beta}$ (denoted $\overline{\alpha}\overline{\beta}$) is a computation iff states $\overline{\alpha}_{n-1}$ and $\overline{\beta}_0$ meet the constraints of Definition 2.1.7. Otherwise, the result of concatenation is null. If Γ and Ψ are two sets containing finite and finite/infinte computations respectively, then $\Gamma\Psi = \{\overline{\alpha}\overline{\beta} \mid (\overline{\alpha} \in \Gamma) \wedge (\overline{\beta} \in \Psi)\}$.

We now define special types of programs based on satisfaction of read/write restrictions and timing constraints. Intuitively, a centralized program is one that is *constrained* by no read/write restrictions, i.e., all processes can read and write all program variables in one atomic step. An *untimed* program is one that has no clock variables, and, hence, no timing constraints. We now precisely define these types of programs.

Definition 2.1.8 (centralized program) We say that a program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$ is a *centralized program* if and only if $\Pi_{\mathcal{P}} = \{p\}$ and $W_p = R_p = V_p$. ∎

Definition 2.1.9 (untimed program) We say that a program $\mathcal{P}$ is an *untimed program* if and only if $X_{\mathcal{P}} = \{\}$. ∎

All other definitions presented in this chapter (e.g., state, transition, computation, etc) can be trivially simplified for special types of programs. For instance, in case of untimed programs, in Definition 2.1.7, states are determined by discrete variables only and the notion of *global time* can be eliminated. In other words, a computation of an untimed program is of the form $\overline{\sigma} = \sigma_0 \rightarrow \sigma_1 \cdots$.

In this dissertation, we only consider the following types of programs:

- (timed) centralized,
- untimed decentralized, and
- untimed centralized,

Throughout the dissertation, for the sake of easier and more intuitive terminology, we refer to the first two above types of programs as *real-time* and *distributed*, respectively. In other words, we only consider distributed programs that have no timing constraints and real-time programs that have no read/write restrictions. The main reason that we do not consider other possible types of programs is due to the complexity bottlenecks that already exist in dealing with the above types of programs. In other words, we believe that more complicated combinations (e.g., distributed real-time programs) can be dealt with, if we have a proper insight into the primitive types.

2.1.1 Timed Guarded Commands

To concisely present the transition predicate (and as a result, computations) of a program, we use timed guarded commands[1]. A *timed guarded command* (also called *timed action*) is of the form:

$$L \;\; :: \;\; g \xrightarrow{\lambda} st;$$

where L is a label, g is a state predicate, st is a statement that describes how the program state is updated, and λ is a set of clock variables that are reset by execution of L. Thus, L denotes the following transition predicate:

$\{(s_0, \nu) \rightarrow (s_1, \nu[\lambda := 0]) \mid (s_0, \nu) \in g$, and

s_1 is obtained by changing s_0 as prescribed by $st\}$.

A *guarded wait command* (also called *delay action*) is of the form:

$$L \;\; :: \;\; g \longrightarrow \textbf{wait};$$

where g identifies the set of states from where delay transitions with arbitrary durations are allowed to be taken as long as g continuously remains true. We note that if a program has several timed guarded commands, the choice of execution of timed guarded commands is non-deterministic, i.e., a guarded command whose guard is true is non-deterministically chosen for execution at each time instance.

Obviously, in untimed programs, the guard of their timed actions (or simply *action*) do not specify any timing constraints and do not reset any clock variables. Moreover, untimed programs have no delay actions. Thus, all actions in untimed programs are simply of the form $L :: g \longrightarrow st$. In the next two subsections, we present two examples to illustrate the concepts of real-time and distributed programs using guarded commands.

2.1.2 Example (Real-Time Traffic Controller)

Consider a one-lane turn-based bridge where cars can travel in only one direction at any time [BK08a]. The bridge is controlled by two traffic signals and each signal changes phase from green to yellow and then to red, based on a set of timing constraints. Moreover, if one signal is red, it will turn green some time after the other signal turns red. A *real-time traffic*

[1]The notion of untimed guarded commands was first introduced by Dijkstra [Dij90].

controller program ($\mathcal{TC}$) for the bridge has two discrete variables to represent the status of the signals, i.e., $V_{\mathcal{TC}} = \{sig_0, sig_1\}$, where sig_0 and sig_1 range over $\{G, Y, R\}$. Thus, at any time, the values of sig_0 and sig_1 show in which direction cars are traveling. Moreover, for each signal, $\mathcal{TC}$ has three timers to change signal phase, i.e., $X_{\mathcal{TC}} = \{x_i, y_i, z_i \mid i = 0, 1\}$. Recall that since $\mathcal{TC}$ is a real-time program, it is centralized (i.e., it only has one process).

The program $\mathcal{TC}$ works as follows. When a signal turns green, it may turn yellow within 10 time units, but not sooner than 1 time unit. Subsequently, the signal may turn red between 1 and 2 time units after it turns yellow. Finally, when the signal is red, it may turn green within 1 time unit after *the other* signal becomes red. Both signals operate identically. Thus, the transition predicate of the real-time traffic controller program can be modeled by the following time guarded commands. For $i \in \{0, 1\}$:

$$
\begin{array}{lll}
\mathcal{TC}1_i :: & (sig_i = G) \wedge (1 \leq x_i \leq 10) & \xrightarrow{\{y_i\}} \quad (sig_i := Y); \\
[] & & \\
\mathcal{TC}2_i :: & (sig_i = Y) \wedge (1 \leq y_i \leq 2) & \xrightarrow{\{z_i\}} \quad (sig_i := R); \\
[] & & \\
\mathcal{TC}3_i :: & (sig_i = R) \wedge (z_j \leq 1) & \xrightarrow{\{x_i\}} \quad (sig_i := G); \\
[] & & \\
\mathcal{TC}4_i :: & ((sig_i = G) \wedge (x_i \leq 10)) \vee & \\
 & ((sig_i = Y) \wedge (y_i \leq 2)) \vee & \\
 & ((sig_i = R) \wedge (z_j \leq 1)) & \longrightarrow \quad \mathbf{wait};
\end{array}
$$

where $j = (i + 1) \mod 2$. Notice that the guard of $\mathcal{TC}3_i$ depends on z timer of signal j. For simplicity, we assume that once a traffic light turns green, all cars from the opposite direction have already left the bridge.

One possible set of initial states for $\mathcal{TC}$ is as follows:

$$
\begin{array}{ll}
I_{\mathcal{TC}} = \{\sigma \mid & (\forall k \in \{0, 1\} \ : \ x_k(\sigma) = y_k(\sigma) = 0) \ \wedge \\
 & (\exists i \in \{0, 1\} \ : \ ((sig_i(\sigma) = G) \ \wedge \ (z_i(\sigma) > 1) \ \wedge \ (sig_{(i+1) \mod 2}(\sigma) = R)))\}.
\end{array}
$$

In other words, $I_{\mathcal{TC}}$ is the set of states where one signal is green, the other one is red and is not allowed to immediately turn green, and all x and y timers are reset.

2.1.3 Example (Distributed Byzantine Agreement)

The *Byzantine agreement* problem was first introduced by Lamport, Shostok, and Pease [LSP82]. The canonical version of the program (denoted $\mathcal{BA}$) consists of a *general*, say g, and three (or more) *non-general* processes: j, k, and l. Since the general process only provides a decision, it is modeled implicitly by two variables. Thus, $\Pi_{\mathcal{BA}} = \{j, k, l\}$. Each non-general process of $\mathcal{BA}$ and the general maintains a decision variable d; for the general, the decision can be either 0 or 1, and for the non-general processes, the decision can be 0, 1, or $\perp$, where the value $\perp$ denotes that the corresponding process has not yet received the decision from the general. Each non-general process also maintains a Boolean variable f that denotes whether or not that process has finalized its decision. For each process, a Boolean variable b shows whether or not the process is Byzantine. Thus, the state space of each process is obtained by discrete variables in

$$
\begin{aligned}
V_{\mathcal{BA}} = {} & \{d.g, d.j, d.k, d.l\} \cup & & \text{(decision variables)} \\
& \{f.j, f.k, f.l\} \cup & & \text{(finalized?)} \\
& \{b.g, b.j, b.k, b.l\}. & & \text{(Byzantine?)}
\end{aligned}
$$

The sets of variables that a non-general processes, say j, is allowed to read and write are $R_j = \{b.j, d.j, f.j, d.k, d.l, d.g\}$ and $W_j = \{d.j, f.j\}$, respectively. The read/write restrictions of processes k and l can be symmetrically instantiated in the same fashion.

The *fault-intolerant* version of $\mathcal{BA}$ works as follows. Each non-general process copies the decision from the general and then finalizes (outputs) that decision, provided it is non-Byzantine. Thus, the transition predicate of a non-general process, say j, is specified by the following two actions:

$$
\begin{array}{l}
\mathcal{BA}1_j \ :: \ (d.j = \perp) \ \wedge \ (f.j = \mathit{false}) \ \wedge \ (b.j = \mathit{false}) \quad \longrightarrow \quad d.j := d.g; \\
[] \\
\mathcal{BA}2_j \ :: \ (d.j \neq \perp) \ \wedge \ (f.j = \mathit{false}) \ \wedge \ (b.j = \mathit{false}) \quad \longrightarrow \quad f.j := \mathit{true};
\end{array}
$$

The actions of processes k and l can be symmetrically instantiated in the same fashion.

The set of initial states of $\mathcal{BA}$ is as follows:

$$I_{\mathcal{BA}} = \{\sigma \mid \forall p \in \{j, k, l\} \ : \ ((d.p(\sigma) = \perp) \ \wedge \ (f.p(\sigma) = \mathit{false}) \ \wedge \ (b.p(\sigma) = \mathit{false})\}.$$

In other words, all non-general processes are non-Byzantine, undecided, and, therefore, their decisions are not finalized. Recall that since $\mathcal{BA}$ is distributed, it is untimed.

2.2 Specifications

In this section, we formally present the concept of specifications and define what it means for a program to refine a specification.

Definition 2.2.1 (specification) A *specification* (or *property*), denoted *SPEC*, is a set of infinite computations of the form $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ where σ_i is a state, $i \in \mathbb{Z}_{\geq 0}$, and the sequence $\tau_0\tau_1\cdots$ meets monotonicity, divergence, and time consistency. ∎

Assumption 2.2.2 Since we use specifications to reason about the correctness of a program, we assume that the state space of a specification is identical to the state space of the program under consideration. ∎

Definition 2.2.3 (refines) Let $\mathcal{P} = \langle \Pi_\mathcal{P}, I_\mathcal{P} \rangle$ be a program and *SPEC* be a specification. We write $\mathcal{P} \models SPEC$ and say that $\mathcal{P}$ *refines SPEC* iff every computation of $\mathcal{P}$ that starts from a state in $I_\mathcal{P}$ is in *SPEC*. ∎

Following Alpern and Schneider [AS85] and Henzinger [Hen92], the specifications considered in this dissertation are an intersection of a *safety* specification and a *liveness* specification defined next. Intuitively, the safety specification of *SPEC* specifies that *nothing bad* should happen in a computation. Safety specifications are often modeled by set of *bad prefixes*. The liveness specification specifies that *something good* must eventually happen.

Definition 2.2.4 (safety specification) A safety specification of *SPEC* is a set of computations that meets the following condition: for each infinite computation $\overline{\sigma}$ that is not in that set, there exists prefix $\overline{\alpha}$ of $\overline{\sigma}$, such that for all infinite computations of the form $\overline{\alpha}\overline{\beta}$, $\overline{\beta}$ is not in that set as well.

Definition 2.2.5 (liveness specifications) A liveness specification of *SPEC* is a set of computations that meets the following condition: for each computation prefix $\overline{\alpha}$, there exists an infinite computation $\overline{\beta}$ such that $\overline{\alpha}\overline{\beta} \in SPEC$. ∎

2.2.1 Example

Real-time Traffic Controller

In the context of our real-time traffic controller, a bad thing that can happen is the case where both signals are *not* red at the same time. Obviously, such a case may be catastrophic on the bridge. Following Definition 2.2.4, the safety specification of $\mathcal{TC}$ can be characterized by the set of *bad transitions* (i.e., bad prefixes of length 2) where both signals are not red in their target states:

$$SPEC_{bt_{\mathcal{TC}}} = \{(\sigma_0, \sigma_1) \mid (sig_0(\sigma_1) \neq R) \ \wedge \ (sig_1(\sigma_1) \neq R)\}.$$

Distributed Byzantine Agreement

In the context of the Byzantine agreement problem, the safety specification of $\mathcal{BA}$ requires *validity*, *agreement*, and *non-enforcement*:

- *Validity* requires that if the general is non-Byzantine, then the final decision of a non-Byzantine process must be the same as that of the general.
- *Agreement* requires that the final decision of any two non-Byzantine processes must be equal.
- *Persistency* requires that once a non-Byzantine process finalizes (outputs) its decision, it cannot change it.

Thus, the following transition predicate characterizes the safety specification of $\mathcal{BA}$:

$$\begin{array}{ll}
SPEC_{bt_{\mathcal{BA}}} = \{(\sigma_0, \sigma_1) \mid & \\
\textit{(validity)} & (\exists p :: \neg b.g(\sigma_1) \ \wedge \ \neg b.p(\sigma_1) \ \wedge \ (d.p(\sigma_1) \neq \bot) \ \wedge \ f.p(\sigma_1) \wedge \\
& \quad (d.p(\sigma_1) \neq d.g(\sigma_1))) \ \vee \\
\textit{(agreement)} & (\exists p, q :: \neg b.p(\sigma_1) \ \wedge \ \neg b.q(\sigma_1) \ \wedge \ f.p(\sigma_1) \ \wedge \ f.q(\sigma_1) \wedge \\
& \quad (d.p(\sigma_1) \neq \bot) \ \wedge \ (d.q(\sigma_1) \neq \bot) \ \wedge \ (d.p(\sigma_1) \neq d.q(\sigma_1))) \ \vee \\
\textit{(Persistency)} & (\exists p :: \neg b.p(\sigma_0) \ \wedge \ \neg b.p(\sigma_1) \ \wedge \ f.p(\sigma_0) \wedge \\
& \quad ((d.p(\sigma_0) \neq d.p(\sigma_1)) \ \vee \ (f.p(\sigma_0) \neq f.p(\sigma_1))))\},
\end{array}$$

where p and q range over non-general processes. In this context, an example of a liveness specification can be "all non-general non-Byzantine processes eventually reach a decision".

2.3 Region Graphs

Real-time programs can be analyzed with the help of an equivalence relation of finite index on the set of states [AD94]. Given a real-time program $\mathcal{P}$, for each clock variable $x \in X$, let c_x be the largest constant in the clock constraints of $T_\mathcal{P}$ that involve x, where $c_x = 0$ if x does not occur in any clock constraint of $\mathcal{P}$. We say that two clock valuations ν, μ are *clock equivalent* if:

1. for all $x \in X$, either $\lfloor \nu(x) \rfloor = \lfloor \mu(x) \rfloor$ or both $\nu(x), \mu(x) > c_x$,

2. the ordering of the fractional parts of the clock variables in the set $\{x \in X \mid \nu(x) < c_x\}$ is the same in μ and ν, and

3. for all $x \in X$, where $\nu(x) < c_x$, the clock value $\nu(x)$ is an integer if and only if $\mu(x)$ is an integer.

A *clock region* ρ is a clock equivalence class. Two states (s_0, ν_0) and (s_1, ν_1) are region equivalent, written $(s_0, \nu_0) \equiv (s_1, \nu_1)$, if (1) $s_0 = s_1$, and (2) ν_0 and ν_1 are clock equivalent. A *region* $r = (s, \rho)$ is an equivalence class with respect to $\equiv$, where s is a location and ρ is a clock region. We say that a clock region β is a *time-successor* of a clock region α iff for each $\nu \in \alpha$, there exists $\tau \in \mathbb{R}_{\geq 0}$, such that $\nu + \tau \in \beta$, and $\nu + \tau' \in \alpha \cup \beta$ for all $\tau' < \tau$. We call a region (s, ρ) a *boundary region*, if for each $\nu \in \rho$ and for any $\tau \in \mathbb{R}_{\geq 0}$, ν and $\nu + \tau$ are not equivalent. A region is *open*, if it is not a boundary region. A region (s, ρ) is called an *end region*, if $\nu(x) > c_x$ for all $\nu \in \rho$ and for all clock variables $x \in X$.

Using the region equivalence relation, we construct the *region graph* of a program $\mathcal{P} = \langle \Pi_\mathcal{P}, I_\mathcal{P} \rangle$ (denoted $R(\mathcal{P}) = \langle \Pi^r_\mathcal{P}, I^r_\mathcal{P} \rangle$) as follows. Vertices of $R(\mathcal{P})$ (denoted $\mathcal{S}^r_\mathcal{P}$) are regions obtained from state space of $\mathcal{P}$. Edges of $R(\mathcal{P})$ (denoted $T^r_\mathcal{P}$) are of the form $(s_0, \rho_0) \rightarrow (s_1, \rho_1)$ iff for some clock valuations $\nu_0 \in \rho_0$ and $\nu_1 \in \rho_1$, $(s_0, \nu_0) \rightarrow (s_1, \nu_1)$ is a transition in $T_\mathcal{P}$. Obviously, the transition predicate of each process p in $\Pi_\mathcal{P}$ has a respective set T^r_p of edges. Thus, $T^r_\mathcal{P} = \bigcup_{p \in \Pi_\mathcal{P}} T^r_p$. A *region predicate* U^r with respect to a state predicate U is defined by $U^r = \{(s, \rho) \mid \exists (s, \nu) : ((s, \nu) \in U \ \wedge \ \nu \in \rho)\}$. Likewise, the region predicate with respect to initial states $I_\mathcal{P}$ of a program $\mathcal{P}$ is called *initial regions* (denoted $I^r_\mathcal{P}$).

Similar to the notion of deadlock states, for a program $\mathcal{P} = \langle \Pi_\mathcal{P}, I_\mathcal{P} \rangle$, we say that a region (s_0, ρ_0) of $R(\mathcal{P})$ is a *deadlock region* iff for all regions $(s_1, \rho_1) \in \mathcal{S}^r_\mathcal{P}$, there does not exist an edge of the form $(s_0, \rho_0) \rightarrow (s_1, \rho_1)$ in $T^r_\mathcal{P}$.

We note that a region graph is a time-abstract bisimulation of the corresponding real-time program [AD94]. In our revision algorithms in Chapters 6, 8, and 9, we transform a real-time program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$ into its corresponding region graph $R(\mathcal{P})$ by invoking the procedure $\mathsf{ConstructRegionGraph}$ as a black box. We also let this procedure take state predicates and transition predicates in $\mathcal{P}$ and return the corresponding region predicates and sets of edges in $R(\mathcal{P})$. Likewise, we transform a region graph $R(\mathcal{P})$ back into a real-time program by invoking the procedure $\mathsf{ConstructRealTimeProgram}$. We note that the above recipe for constructing region graphs involves an exponential blow-up in the number of clocks and also in the magnitude of clock variables. Thus, when we analyze complexity of algorithms in the size of region graphs, the reader should automatically consider this construction complexity as well.

Part II

Revising Programs in Closed Systems

In this part of the dissertation, we present our results on automated revision of programs in *closed systems*. In a closed system, programs do not interact with the environment in any ways. More specifically, in closed systems the state of a program is not affected by environment or uncontrollable fault transitions. Thus, we intuitively interpret the revision problem as follows:

> Given a program $\mathcal{P}$ and a (new) specification $SPEC_n$, where $\mathcal{P}$ does not refine $SPEC_n$, the problem is whether or not it is possible to automatically revise $\mathcal{P}$ inside the state space of $\mathcal{P}$ such that the revised program refines $SPEC_n$ while it continues to satisfy its existing specification $SPEC_e$.

This part is organized as follows. First, we present the type of safety and liveness properties that we consider in this part and we formally state the problem of revising programs in closed systems in Chapter 3. Following the problem statement, in Chapters 4, 5, and 6, we present our results on the complexity of revising untimed centralized, distributed, and real-time programs, respectively.

Chapter 3

The Revision Problem in Closed Systems

In this chapter, we formalize the problem of revising programs in closed systems. This chapter is organized as follows. Section 3.1 is dedicated to present the type of specifications that we consider in revising programs in closed systems. Then, in Section 3.2, we formally state the revision problem.

3.1 Basic Concepts

In the context of revising programs in closed systems, we concentrate on UNITY properties introduced by Chandy and Misra [CM88] and real-time UNITY properties introduced by Carruth [Car94]. The reason for our specific focus on UNITY properties is that they are known to be highly expressive in specifying a large class of programs.

Definition 3.1.1 (real-time UNITY properties) Let P and Q be arbitrary state predicates.

- (Bounded-time unless) An infinite timed state sequence $\overline{\sigma} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ refines '$P \ \mathsf{unless}_\delta \ Q$' iff $\forall i \geq 0 : ((\sigma_i \in (P \cap \neg Q)) \Rightarrow \forall j > i \mid (\tau_j - \tau_i \leq \delta) : (\sigma_j \in (P \cup Q)))$. Intuitively, if P holds at any σ_i, then for all $j > i$ such that $\tau_j - \tau_i \leq \delta$ either (1) Q does not hold in σ_j and P is true, or (2) Q becomes true in σ_j and P holds at least until Q becomes true. After δ time units there is no requirement on P and Q.

- (Bounded-time leads-to) An infinite timed state sequence $\overline{\sigma} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ refines 'P leads-to$_\delta$ Q' (denoted $P \mapsto_\delta Q$) iff for all $i \geq 0$, if $\sigma_i \in P$, then there exists j, $j \geq i$, such that (1) $\sigma_j \in Q$, and (2) $\tau_j - \tau_i \leq \delta$. Intuitively, it is always the case that a state in P is followed by a state in Q within δ time units. ∎

We now present untimed UNITY properties by abstracting the notion of time from Definition 3.1.1. This abstraction results in transformation of bounded-time leads-to property from a safety property into a liveness property. Precisely, Chandy and Misra classify untimed UNITY properties in two broad categories of *safety* and *progress* properties defined next.

Definition 3.1.2 (untimed UNITY safety properties) Let P and Q be arbitrary state predicates.

- (Unless) An infinite sequence of states $\overline{\sigma} = \sigma_0 \rightarrow \sigma_1 \cdots$ refines 'P unless Q' iff $\forall i \geq 0 : (\sigma_i \in (P \cap \neg Q)) \Rightarrow (\sigma_{i+1} \in (P \cup Q))$. Intuitively, if P holds in a state of $\overline{\sigma}$, then either (1) Q never holds in $\overline{\sigma}$ and P is continuously true, or (2) Q becomes true and P holds at least until Q becomes true.

- (Stable) An infinite sequence of states $\overline{\sigma} = \sigma_0 \rightarrow \sigma_1 \cdots$ refines 'stable P' iff $\overline{\sigma}$ satisfies P unless *false*. Intuitively, P is stable iff once it becomes true, it remains true forever.

- (Invariant) An infinite sequence of states $\overline{\sigma} = \sigma_0 \rightarrow \sigma_1 \cdots$ refines 'invariant P' iff $\sigma_0 \in P$ and $\overline{\sigma}$ satisfies stable P. An invariant property always holds. ∎

Definition 3.1.3 (untimed UNITY progress properties) Let P and Q be arbitrary state predicates.

- (Leads-to) An infinite sequence of states $\overline{\sigma} = \sigma_0 \rightarrow \sigma_1 \cdots$ refines 'P leads-to Q' (denoted $P \mapsto Q$) iff $(\forall i \geq 0 : (\sigma_i \in P) \Rightarrow (\exists j \geq i : \sigma_j \in Q))$. In other words, if P holds in a state σ_i, $i \geq 0$, then there exists a state σ_j in $\overline{\sigma}$, $i \leq j$, such that Q holds in σ_j.

- (Ensures) An infinite sequence of states $\overline{\sigma} = \sigma_0 \rightarrow \sigma_1 \cdots$ refines 'P ensures Q' iff for all i, $i \geq 0$, if $P \cap \neg Q$ is true in state σ_i, then (1) $\sigma_{i+1} \in (P \cup Q)$, and (2)

$\exists j \geq i : \sigma_j \in Q$. In other words, if P becomes true in σ_i, there exists a state σ_j where Q eventually becomes true and P remains true everywhere in between σ_i and σ_j. ∎

There exist some small differences between our formal framework and that in standard UNITY. First, in our formal framework, unlike the standard UNITY which assumes *interleaved fairness*, we assume that all program computations are unfair. This assumption is necessary in dealing with polynomial-time addition of UNITY progress properties to programs. Secondly, our definition of ensures is slightly different from that in [CM88]. In Chandy and Misra's definition, (P ensures Q) implies (1) P leads-to Q, (2) P unless Q and (3) there is at least one action that always establishes Q whenever it is executed in a state where P is true and Q is false. Since, we do not model actions explicitly in our work, we have removed the third requirement. Finally, as described in Chapter 2, our focus is on programs with a finite set of discrete variables. Thus, in the context of untimed programs, all programs have *finite* state space.

Definition 3.1.1 lacks two types or properties as compared to the definition of untimed UNITY properties. However, one can easily instantiate the definition of bounded-time stable and bounded-time ensures properties in the obvious way. The definition of invariant is not time-related and, therefore, remains the same.

Assumption 3.1.4 In Part II of the dissertation, we add the following constraint to Definition 2.1.7. Let $\mathcal{P} = \langle \Pi_\mathcal{P}, I_\mathcal{P} \rangle$ be a program. For all computations $\overline{\sigma} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ of $\mathcal{P}$, we require that $\sigma_0 \in I_\mathcal{P}$. ∎

Given Definitions 3.1.1, 3.1.2, and 3.1.3, it is straightforward to precisely define the notion of *specifications*. A UNITY specification *SPEC* is the conjunction $\bigwedge_{i=1}^{n} \mathcal{L}_i$ where (depending upon the context) each $\mathcal{L}_i$ is either a real-time UNITY property, or, an untimed UNITY (safety or progress) property.

In the context of refinement of a UNITY specification by a program, following Definition 2.2.3, we say that a program $\mathcal{P} = \langle \Pi_\mathcal{P}, I_\mathcal{P} \rangle$ refines a UNITY specification $SPEC = \bigwedge_{i=1}^{n} \mathcal{L}_i$ if and only if $\mathcal{P}$ refines $\mathcal{L}_i$ for all i, $1 \leq i \leq n$. In other words, all computations of $\mathcal{P}$ refine all UNITY properties in *SPEC*.

Concise Representation of Untimed Unity Safety Properties

Observe that the untimed UNITY safety properties can be characterized in terms of a set of *bad transitions* that should never occur in a program computation. For example, stable P requires that a transition, say (σ_0, σ_1), where $\sigma_0 \in P$ and $\sigma_1 \notin P$, should never occur in any computation of a program that refines stable P. Hence, for simplicity, in Part II of the dissertation, when dealing with safety UNITY properties of a program $\mathcal{P}$, we assume that they are represented by a transition predicate $\mathcal{B} \subseteq \mathcal{S}_\mathcal{P} \times \mathcal{S}_\mathcal{P}$ whose transitions should never occur in any computation. Examples of such "bad transition" were presented in Subsection 2.2.1.

3.2 Problem Statement

Given are a program $\mathcal{P} = \langle \Pi_\mathcal{P}, I_\mathcal{P} \rangle$ and a (new) UNITY specification $SPEC_n$. Our goal is to devise an automated method which revises $\mathcal{P}$ so that the revised program (denoted $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, I_{\mathcal{P}'} \rangle$) (1) refines $SPEC_n$, and (2) continues refining its existing UNITY specification $SPEC_e$, where $SPEC_e$ is unknown. Thus, during the revision, we only want to reuse the correctness of $\mathcal{P}$ with respect to $SPEC_e$ so that the correctness of $\mathcal{P}'$ with respect to $SPEC_e$ is derived from '$\mathcal{P}$ refines $SPEC_e$'.

Intuitively, in order to ensure that the revised program $\mathcal{P}'$ continues refining the existing specification $SPEC_e$, we constrain the revision problem so that the set of computations of $\mathcal{P}'$ is a subset of the set of computations of $\mathcal{P}$. In this sense, since UNITY properties are not existentially quantified (unlike in CTL [Eme90]), we are guaranteed that all computations of $\mathcal{P}'$ satisfy the UNITY properties that participate in $SPEC_e$.

Notation. Let $\mathcal{P}$ be a program. Let Y be a subset of the set of clock variables of $\mathcal{P}$, i.e., $Y \subseteq X_\mathcal{P}$. We denote the program obtained by removing the clock variables in Y from $X_\mathcal{P}$ by $\mathcal{P} \backslash Y$. Obviously, no state and transition predicate of $\mathcal{P} \backslash Y$ depend on the value of variables in Y.

Now, we formally identify constraints on state space of $\mathcal{P}'$ (denoted $\mathcal{S}_{\mathcal{P}'}$), initial states of $\mathcal{P}'$, $I_{\mathcal{P}'}$, and transition predicate of $\mathcal{P}'$ (denoted $T_{\mathcal{P}'}$). Observe that:

- if $\mathcal{S}_{\mathcal{P}'}$ contains states that are not in $\mathcal{S}_\mathcal{P}$, there is no guarantee that the correctness of $\mathcal{P}$ with respect to $SPEC_e$ can be reused to ensure that $\mathcal{P}'$ refines $SPEC_e$. Also, since $\mathcal{S}_\mathcal{P}$ denotes the set of all states (not just reachable states) of $\mathcal{P}$, removing states from $\mathcal{S}_\mathcal{P}$ is not advantageous. However, since meeting new timing constraints requires time predictability, we let revision methods incorporate a finite set X_n of new clock variables in order to keep track of time only. Thus, in the revision problem, we require that $\mathcal{S}_{\mathcal{P}'\setminus X_n} = \mathcal{S}_{\mathcal{P}}$.

- Likewise, $I_{\mathcal{P}'}$ should not have any states that were not there in $I_{\mathcal{P}}$. Moreover, since $I_{\mathcal{P}}$ denotes the set of all initial states of $\mathcal{P}$, we should preserve them during the revision. Thus, we require that $I_{\mathcal{P}'\setminus X_n} = I_{\mathcal{P}}$.

- Finally, we require that $T_{\mathcal{P}'\setminus X_n}$ should be a subset of $T_{\mathcal{P}}$. Note that not all transitions in $T_{\mathcal{P}}$ may be preserved in $T_{\mathcal{P}'}$. Hence, we must ensure that $\mathcal{P}'$ does not deadlock. Based on Definitions 2.2.1 and 2.2.3, if

  1. $T_{\mathcal{P}'\setminus X_n} \subseteq T_{\mathcal{P}}$,
  2. $\mathcal{P}'$ does not deadlock, and
  3. $\mathcal{P}$ refines $SPEC_e$,

  then $\mathcal{P}'$ also refines $SPEC_e$.

Thus, the *revision problem* is formally defined as follows:

Problem Statement 3.2.1 Given a program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$ and a UNITY specification $SPEC_n$, identify $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, I_{\mathcal{P}'} \rangle$ such that:

$(C1)$ $\mathcal{S}_{\mathcal{P}'\setminus X_n} = \mathcal{S}_{\mathcal{P}}$,

$(C2)$ $I_{\mathcal{P}'\setminus X_n} = I_{\mathcal{P}}$,

$(C3)$ $T_{\mathcal{P}'\setminus X_n} \subseteq T_{\mathcal{P}}$, and

$(C4)$ $\mathcal{P}'$ refines $SPEC_n$. ∎

Note that the requirement of deadlock freedom is not explicitly specified in the above problem statement, as it follows from '$\mathcal{P}'$ refines $SPEC_n$'. Throughout Part II of the dissertation, we use '*revision* of $\mathcal{P}$ with respect to a specification $SPEC_n$ (or property $\mathcal{L}$)' and '*addition* of $SPEC_n$ (respectively, $\mathcal{L}$) to $\mathcal{P}$' interchangeably. In Chapters 4, 5, and 6, we present the complexity of revision methods that solve Problem Statement 3.2.1 with respect to untimed centralized, distributed, and real-time programs, respectively for different types of UNITY properties. In this context, we address two important criteria with respect to revision algorithms: *soundness* and *completeness*.

Definition 3.2.2 **(soundness)** We say that a revision algorithm is *sound* iff its output meets the constraints of Problem Statement 3.2.1. ∎

Definition 3.2.3 **(completeness)** We say that a revision algorithm is *complete* iff it finds a solution to Problem Statement 3.2.1 iff there exists one. ∎

Chapter 4

Revising Untimed Centralized Programs

In this chapter, we present our contributions on automated revision of untimed centralized programs in closed systems. Although such programs are neither distributed nor real-time, identifying the complexity of their revision is of interest in the sense that a hardness result automatically determines a lower bound on the complexity of the corresponding problem in the distributed and timed settings.

This chapter is organized as follows. In Section 4.1, we present a polynomial-time sound and complete algorithm for adding a single unbounded progress UNITY property along with a conjunction of safety UNITY properties to an untimed centralized program. Then, in Section 4.2, we show that while it is possible to add a single progress property in polynomial-time, the problem of adding two or more progress properties is significantly more difficult, i.e., it is NP-complete. Finally, in Section 4.3, we show that addition of even one leads-to property to an untimed centralized program while retaining maximum number of transitions is NP-complete.

4.1 Adding a Single Progress and Multiple Safety Properties

In this section, we present a simple solution for Problem Statement 3.2.1 where the new specification $SPEC_n$ is a conjunction of a single leads-to property and multiple safety properties. We note that the goal of our algorithm is simply to illustrate the feasibility of a polynomial-time solution. Hence, although our algorithm in this section can be modified to reduce the complexity further, we have chosen to present a simple (and not so efficient) solution (see Algorithm 4.1).

Let $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$ be a program and specification $SPEC_n \equiv \mathcal{B} \wedge \mathcal{L}$, where $\mathcal{B}$ represents the conjunction of a set of safety properties and $\mathcal{L} \equiv (P \mapsto Q)$ for arbitrary state predicates P and Q in $\mathcal{S}_{\mathcal{P}}$. In order to guarantee that the revised program $\mathcal{P}'$ satisfies $\mathcal{B}$ (i.e., $\mathcal{P}'$ never executes a transition in the set of bad transitions $\mathcal{B}$), we simply remove all transitions in $\mathcal{B}$ from $T_{\mathcal{P}}$ (Step 1).

In order to add the leads-to property $\mathcal{L} \equiv (P \mapsto Q)$ to $\mathcal{P}$, we need to guarantee that any computation of $\mathcal{P}'$ that reaches a state in P will eventually reach a state in Q. Towards this end, we rank all states σ in $\mathcal{S}_{\mathcal{P}}$ based on the length of the shortest computation prefix of $\mathcal{P}$ from σ to a state in Q (Step 2). In such a ranking, if no state of Q is reachable from σ then the rank of σ will be *infinity*. Also, the rank of states in Q is zero. There exist two obstacles in guaranteeing the reachability of Q from P: (1) deadlock states reachable from P, and (2) cycles reachable from P in which computations of $\mathcal{P}'$ may be trapped forever. In addition to possible existing deadlock states in $\mathcal{P}$, our algorithm may also introduce deadlock states by (i) removing safety-violating transitions (Step 1), and (ii) making infinity-ranked states in P unreachable in Step 4.

Regarding deadlock states, our approach is to make them unreachable (Steps 5-12). Such removal of transitions may introduce new deadlock states that are removed in the *while* loop. If the removal of deadlock states culminates in making an initial state deadlocked then $(P \mapsto Q)$ cannot be added to $\mathcal{P}$. Otherwise, we again rank all states (Step 13) as we might have removed some deadlock states in Q, and consequently, created new infinity-ranked states. We repeat the above steps until no reachable state in P has the rank infinity. At this point (end of the repeat-until loop), there is a path from each state in P to a state in Q. However, there may exist a computation prefix $\langle \sigma_0, \sigma_1, \cdots, \sigma_n \rangle$ such that (1) $\sigma_0 \in P$, (2) $\sigma_n \in Q$, (3) for all $i \in \{1..n-1\} : \sigma_i \notin Q$, and (4) $\exists j \in \{2..n-1\}$ where σ_j is on a cycle.

To deal with such cycles, we retain transitions from high-ranked states to low-ranked states (Step 15). In particular, if $\mathrm{Rank}(\sigma_0) \leq \mathrm{Rank}(\sigma_1)$ then it means there exists a computation prefix of shorter or equal length from σ_0 to Q as compared to the computation prefix from σ_1 to Q. Thus, removing (σ_0, σ_1) will not make σ_0 deadlocked. Notice that in Step 15, transitions of the form (σ_0, σ_1), where $\mathrm{Rank}(\sigma_0) = \infty$ and $\mathrm{Rank}(\sigma_1) = \infty$, are not removed. Also, we ensure that no transitions that originate from Q is removed. Hence, computations in which neither predicates P and Q are reached will not be affected.

Remark. We note that since ensures can be expressed as a conjunction of an unless property and a leads-to property, our algorithm is able to add an ensures property as well.

Algorithm 4.1 Add_UNITY

Input: untime program $\langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$, leads-to property $P \mapsto Q$, and safety specification $\mathcal{B}$.

Output: revised program $\langle \Pi_{\mathcal{P}'}, I_{\mathcal{P}'} \rangle$.

1: $T_{\mathcal{P}_1} := T_{\mathcal{P}} - \{(\sigma_0, \sigma_1) \mid (\sigma_0, \sigma_1) \in \mathcal{B}\}$;

2: $\forall \sigma \in \mathcal{S}_{\mathcal{P}} : \text{Rank}(\sigma) =$ the length of the shortest computation prefix of $T_{\mathcal{P}_1}$ that starts from σ and ends in a state in Q;

{$\text{Rank}(\sigma) = \infty$ *means* Q *is not reachable from* σ.}

3: $T_{\mathcal{P}_1} := T_{\mathcal{P}_1} - \{(\sigma_0, \sigma_1) \mid (\sigma_1 \in P) \wedge \text{Rank}(\sigma_1) = \infty\}$;

$(\exists \sigma_0 \in \mathcal{S}_{\mathcal{P}} : (\forall \sigma_1 \in \mathcal{S}_{\mathcal{P}} : (\sigma_0, \sigma_1) \notin T_{\mathcal{P}_1}))$

$(\sigma_0 \notin I_{\mathcal{P}})$

4: $T_{\mathcal{P}_1} := T_{\mathcal{P}_1} - \{(\sigma, \sigma_0) \mid (\sigma, \sigma_0) \in T_{\mathcal{P}_1}\}$;

5: `declare that the addition is not possible`;

6: **exit**();

7: $\forall \sigma \in \mathcal{S}_{\mathcal{P}} : \text{Rank}(\sigma) =$ the length of the shortest computation prefix of $T_{\mathcal{P}_1}$ that starts from σ and ends in a state in Q;

$(\forall \sigma \mid (\sigma \in P) \wedge (\sigma \text{ is reachable from } I_{\mathcal{P}} \text{ using } T_{\mathcal{P}_1}) : \text{Rank}(\sigma) \neq \infty)$

8: $T_{\mathcal{P}_1} - \{(\sigma_0, \sigma_1) \mid (\text{Rank}(\sigma_0) > 0) \wedge (\text{Rank}(\sigma_0) \neq \infty) \wedge (\text{Rank}(\sigma_0) \leq \text{Rank}(\sigma_1))\}$;

Theorem 4.1.1 *The* Add_UNITY *algorithm is sound and complete.*

Proof. Since Add_UNITY does not add any new states to $\mathcal{S}_{\mathcal{P}}$, we have $\mathcal{S}_{\mathcal{P}'} = \mathcal{S}_{\mathcal{P}}$. Likewise, Add_UNITY does not remove (respectively, introduce) any initial states; we have $I_{\mathcal{P}'} = I_{\mathcal{P}}$. The Add_UNITY algorithm only updates $T_{\mathcal{P}}$ by excluding some transitions from $T_{\mathcal{P}}$ in Steps 1, 4, 7, and 15. It follows that $T_{\mathcal{P}'} \subseteq T_{\mathcal{P}}$. By construction, if the Add_UNITY algorithm generates a program $T_{\mathcal{P}'}$ in Step 15 then reachability from P to Q is guaranteed in $\mathcal{P}'$. Thus, $\mathcal{P}'$ meets all the requirements of Problem Statement 3.2.1.

We now show that the algorithm is complete. Note that any transition removed in Add_UNITY (in Steps 1, 4, and 7) must be removed in any program that meets the requirements of Problem Statement 3.2.1. Hence, if failure is declared (in Step 9), there exists no solution to Problem Statement 3.2.1.

Recall that a program $\mathcal{P}$ satisfies a specification $SPEC$ iff all computations of $\mathcal{P}$ are in $SPEC$. Hence, a subset of computations of $\mathcal{P}$ satisfies *spec* as well. In the context of the algorithm Add_UNITY, although it excludes some computations, since it ensures that all computations are infinite (by removing deadlock regions), it continues to satisfy its old UNITY specification. Note, however, that the same result cannot be implied for specification languages that have existential quantification features such as branching-time temporal logics TCTL.

Theorem 4.1.2 *The complexity of* Add_UNITY *algorithm is polynomial-time in* $\mathcal{S}_\mathcal{P}$. ∎

Remark. We would like to note that soundness and completeness of Add_UNITY are preserved for the case where the revised program is allowed to have a subset of initial states of the original program. For such a case, the algorithm would fail only if all initial states are removed.

4.1.1 Example: Readers-Writers Program

In this section, we illustrate the application of the Add_UNITY algorithm in local redesign of a program for the readers-writers problem [CM88]. As usual, we use Dijkstra's guarded commands (*actions*) as a shorthand for representing the set of program transitions. Recall that a guarded command $g \rightarrow st$ captures the transitions $\{(\sigma_0, \sigma_1) \mid$ the state predicate g is true in σ_0, and σ_1 is obtained by *atomic* execution of statement st in state $s_0\ \}$.

The Readers-Writers Program

In the The readers-writers program (denoted $\mathcal{RW}$), multiple writer processes wait in an infinite external queue to be picked by the program. $\mathcal{RW}$ contains a finite internal queue of size 2 that is managed by a *queue manager* process, which selects writers from an external queue and places them in the internal queue. The selected writer has access to a shared buffer to which other processes have access as well. At any time only one writer is allowed to write the buffer. The reader processes can read the shared buffer. The program has three integer variables $0 \leq nr \leq N$, $0 \leq nw \leq N$, and $0 \leq nq \leq 2$ that are initially 0, where N denotes the total number of processes. Specifically, nr represents the number of readers reading from the buffer, nw represents the number of writers writing the buffer, and nq represents the number of writers waiting in the internal queue. The program contains Boolean variables rd_j $(1 \leq j \leq N)$, and wrq that respectively represent whether or not the reader R_j is reading the buffer, and at least a writer is waiting in the internal queue. The variable wrq is set to *true* by the queue manager when there is a process *waiting* to write and wrq is set to *false* when a process is writing the buffer.

Safety Specification

The safety specification, $\mathcal{B}_{\mathcal{RW}}$, of the program requires that when a writer is writing in the buffer no other process is allowed to access the buffer. However, multiple readers can read the buffer simultaneously:

$$\mathcal{B}_{\mathcal{RW}} = \{(s_0, s_1) \mid (nw(s_1) > 1) \ \vee \ ((nr(s_1) \neq 0) \ \wedge \ nw(s_1) \neq 0))\}$$

The safety specification stipulates that the condition $(nw \leq 1) \ \wedge \ ((nr = 0) \vee (nw = 0))$ must hold in every reachable state. Another representation of the above formula is $0 \leq (N - (nr + N \cdot nw))$. For ease of presentation, we represent the expression $(N-(nr + N \cdot nw))$ with the variable K.

Actions of RW

The actions of the writer processes in the original program are as follows:

$$\begin{array}{llll}
\mathcal{RW}1 :: & (nq > 0) \ \wedge \ (K \geq 3) & \longrightarrow & nw := nw + 1;\ nq := nq - 1;\ wrq := \mathit{false}; \\
\mathcal{RW}2 :: & (nw = 1) & \longrightarrow & nw := nw - 1;
\end{array}$$

When there exists a process ready for writing in the internal queue (i.e., $nq > 0$) and no process is using the buffer (i.e., $K \geq 3$), the program allows the writers to write the common buffer. Thus, the writer process waits until all readers finish their reading activities. When a writer process accesses the buffer, it increments the value of nw, sets the value of wrq to *false*, and decrements the value of nq (see action $\mathcal{RW}1$). This way, the queue manager lets other waiting writers in. When the writer finishes its writing activity in the buffer, it exits by decrementing the value of nw (see action $\mathcal{RW}2$).

The following parameterized actions represent the transitions of the readers as the structures of the readers are symmetric:

$$\mathcal{RW}3_j :: \quad \neg wrq \wedge \neg rd_j \wedge (1 < K) \quad \longrightarrow \quad nr := nr + 1;\ rd_j := true;$$
$$\mathcal{RW}4_j :: \quad rd_j \quad \longrightarrow \quad nr := nr - 1;\ rd_j := false;$$

The condition $K > 1$ holds if no writer process is writing the buffer an at most $N - 1$ readers exist. Thus, if a reader process is not already in reading status and no writer is waiting to write the buffer (see action $\mathcal{RW}3_j$) then the reader can read the buffer. (The original program gives the priority to the writers.) When a reader process j completes its reading activity, it decrements the value of nr and sets rd_j to $false$. Now, we present the action of the queue manager process.

$$\mathcal{RW}5 :: \quad (nq < 2) \quad \longrightarrow \quad nq := nq + 1;\ wrq := true;$$

Once the queue manager selects a waiting writer, it increments the value of nq and sets wrq to $true$ in order to show that a writer is waiting in the internal queue. We consider a version of the $\mathcal{RW}$ program where we have two readers, i.e., $j \in \{0, 1\}$, and one writer (i.e., $N = 3$).

Initial States

Let $\langle nr, nw, nq, rd_0, rd_1, wrq\rangle$ denote the state of the RW program. We consider the set $I_{\mathcal{RW}} = \langle 0, 0, 0, false, false, false\rangle$ for initial state the $\mathcal{RW}$ program.

The Desired Leads-to Property

The initial program satisfies the safety specification $\mathcal{B}_{RW}$, however, no progress is guaranteed. For example, the writer process may wait forever due to alternating access of the readers to the buffer. Readers may also wait forever due to continuous presence of writers in the internal queue. Thus, the desired leads-to property for a reader $j \in \{0, 1\}$, is $(0 \leq K) \mapsto (rd_j)$ and the program should satisfy $(nq > 0 \mapsto (nw = 1))$ to ensure that writers have progress. In this example, we present only the redesign of $\mathcal{RW}$ for the property $(0 \leq K) \mapsto (rd_0)$ for the reader R_0. As such, in the property $P \mapsto Q$ in the input of Add_UNITY, the state predicate P is equal to $0 \leq K$ and the state predicate Q equals to rd_0 (see input parameters of Algorithm 4.1).

Adding Leads-to Using Add_UNITY

We trace the execution of Add_UNITY for the addition of $(0 \leq K) \mapsto (rd_0)$ to the RW program.

- *Step 1.* Since the initial program satisfies its safety specification $\mathcal{B}_{RW}$, Step 1 of the Add_UNITY algorithm would not eliminate any transitions.

- *Step 2.* Rank 0 includes eight reachable states where $rd_0 = true$. These states are as follows: $\langle 1, 0, 0, true, false, false\rangle$, $\langle 1, 0, 1, true, false, true\rangle$, $\langle 2, 0, 0, true, true, false\rangle$, $\langle 1, 0, 2, true, false, true\rangle$, $\langle 2, 0, 1, true, true, true\rangle$, $\langle 2, 0, 2, true, true, true\rangle$, $\langle 2, 0, 1, \ true, \ true, \ false\rangle$, and $\langle 1, 0, 1, true, false, false\rangle$.

  From the initial state $\langle 0, 0, 0, false, false, false\rangle$, the reader $j = 0$ can read the buffer and the program reaches the state $\langle 1, 0, 0, true, false, false\rangle$. Thus, the rank of the initial state is 1. Moreover, the reader R_0 can read the buffer from the states $\langle 1, 0, 0, false, true, false\rangle$, $\langle 1, 0, 1, false, true, false\rangle$, and $\langle 0, 0, 1, false, false, false\rangle$. As a result, the program state changes to a state in Rank 0.

  The states $\langle 0, 1, 0, false, false, false\rangle$ and $\langle 0, 1, 1, false, false, false\rangle$ have Rank 2 as the execution of action $\mathcal{RW}2$ from these states changes the program state to a state in Rank 1. Likewise, the states $\langle 0, 0, 1, false, false, true\rangle$ and $\langle 0, 0, 2, \ false, \ false, \ true\rangle$ get Rank 3. Rank 4 includes $\langle 1, 0, 1, \ false, \ true, \ true\rangle$, $\langle 0, 1, 1, false, false, true\rangle$, $\langle 0, 1, 2, false, false, true\rangle$, and $\langle 1, 0, 2, false, true, true\rangle$.

- *Step 4.* There are no states with rank ∞.

- *Steps 5-12.* Since Step 4 does not remove any transitions, no deadlock states are created and, hence, the algorithm does not enter the while loop.

- *Step 13.* This step results in the same ranking as in Step 2.

- *Step 14.* Since all reachable states, where $0 \leq K$ holds, have a finite rank, the algorithm exits the repeat-until loop.

- *Step 15.* This step removes transitions that start in a low ranking state outside Rank 0 and terminate in a higher rank. For example, the transition $(s_0, s1)$ included in action W_1, where $s_0 = \langle 0, 0, 1, false, false, false\rangle$ and $\langle 0, 1, 0, \ false, \ false, \ false\rangle$, starts in Rank 1 and ends in Rank 2. From s_0, the execution of action QM gets the program to state $\langle 0, 0, 2, false, false, false\rangle$ in Rank 3. Moreover, transitions that form a cycle between the states of the same rank (outside Rank 0) are removed. For instance, the reader $j = 1$ may read the buffer from s_0 and the program reaches the state $s_1 = \langle 1, 0, 1, \mathit{false}, \mathit{true}, \mathit{false} \rangle$. Afterwards, reader $j = 1$ may take the state of the program back to s_0 by executing the action $\mathcal{RW}4_1$, thereby, creating a cycle between s_0 and s_1 in Rank 1.

The Revised Program

After applying the Add_UNITY algorithm on the $\mathcal{RW}$ program for properties $(0 \leq K) \mapsto (rd_0)$, $(0 \leq K) \mapsto (rd_1)$ and subsequently $(nq > 0) \mapsto (nw = 1)$ the final revised program is as follows:

$$\begin{array}{llll}
\mathcal{RW}'1 :: & (wrq) \wedge (K = 3) & \longrightarrow & nw := nw + 1;\ nq := nq - 1;\ wrq := \mathit{false}; \\
\mathcal{RW}'2 : & (\neg wrq) \wedge (K = 0) & \longrightarrow & nw := nw - 1;
\end{array}$$

Intuitively, a waiting writer is allowed to write if no other processes have accessed the buffer (i.e., $(wrq) \wedge (K = 3)$). The value of K is zero only if a writer has accessed the buffer, thereby, enabling the writer to release the buffer. The following parameterized action represents the transitions of the reader processes ($j = 0, 1$). A reader process is allowed to read the buffer if no writer is waiting in the internal queue and at most one reader is reading the buffer (i.e., $K > 1$). The guard of the second action has been strengthened in that a reader is allowed to release the buffer if a writer is waiting for access.

$$\begin{array}{llll}
\mathcal{RW}'3_j :: & (\neg wrq) \wedge \neg rd_j \wedge (1 < K) & \longrightarrow & nr := nr + 1;\ rd_j := \mathit{true}; \\
\mathcal{RW}'4_j :: & rd_j \wedge (wrq) \wedge (K < 3) & \longrightarrow & nr := nr - 1;\ rd_j := \mathit{false};
\end{array}$$

The behavior of the queue manager process is also modified in that a writer is put in the internal buffer if (1) no writer is currently in the internal buffer, (2) no writer is writing the buffer, and (3) exactly two readers are reading the buffer (i.e., $K = 1$).

$$\mathcal{RW}'5 : \quad (nq = 0) \ \wedge \ (K = 1) \ \wedge \ (nw = 0) \quad \longrightarrow \quad nq := nq + 1;\ wrq := \mathit{true};$$

We refer the reader to [EKB05] for another example on revising a mutual exclusion algorithm which originally exhibits starvation.

4.2 Adding Multiple Progress Properties

In this subsection, we focus on addition of a combination of progress properties (i.e., leads-to and/or ensures). In this context, we note that the algorithm Add_UNITY can be applied in a stepwise fashion to add multiple progress properties. However, while such stepwise addition is sound, it is not complete. This is due to the fact that during the addition of the first (for instance, leads-to) property, the transitions removed in the last step (Line 15 in Algorithm 4.1) may cause failure in adding subsequent progress properties.

We consider a special case of the problem of adding multiple progress properties where two eventually properties are added to a given program. The property 'eventually Q' is logically equivalent with '$true \mapsto Q$' (respectively, $true$ ensures Q), i.e., starting from an arbitrary state, the program reaches a state in Q. This property in Linear Temporal Logic (LTL) is denoted by $\Box\Diamond Q$ (called *always eventually* Q). Thus, for an infinite computation, this implies that Q must be reached infinitely often. Since this special case is NP-complete (see Theorem 4.2.1 below), the hardness of adding a combination of two leads-to and ensures properties follows trivially.

Instance. A program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$ and $SPEC_n \equiv \mathcal{L}_1 \wedge \mathcal{L}_2$, where $\mathcal{L}_1 \equiv \Box\Diamond P$ and $\mathcal{L}_2 \equiv \Box\Diamond Q$, and P and Q are two arbitrary state predicates.

The decision problem (2EV). Given the above instance, does there exist a program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, I_{\mathcal{P}'} \rangle$ such that $\mathcal{P}'$ satisfies the constraints of Problem Statement 3.2.1?

To show the complexity of the above decision problem, we reduce the problem of determining whether or not a directed graph has a simple cycle that includes two specific vertices, described next, to the problem of adding two eventually properties.

Cycle Detection in Directed Graphs (CDDG). Given a directed graph $G = \langle V, A \rangle$, where V is a set of vertices and A is a set of arcs, and two vertices, say u and v in V, does there exist a (simple) cycle in G that includes both u and v? The CDDG problem is known to be NP-complete [BJG02].

Theorem 4.2.1 *The problem of adding two eventually properties to an untimed program is NP-complete.*

Proof. Since showing membership to NP is straightforward, we only need to show that the problem is NP-hard. Towards this end, we present a polynomial-time mapping from an instance of CDDG to a corresponding instance of the 2EV problem. Let $G = \langle V, A\rangle$ be a directed graph. We construct $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}}\rangle$ and identify $SPEC_n \equiv \mathcal{L}_1 \wedge \mathcal{L}_2$ as follows:

- $\mathcal{S}_= \{s_x \mid x \in V\}$,
- $T_= \{(s_x, s_y) \mid (x, y) \in A\}$,
- $I_= \{s_u, s_v\}$,
- $\mathcal{L}_1 \equiv \Box\Diamond\{s_u\}$, and $\mathcal{L}_2 \equiv \Box\Diamond\{s_v\}$.

Now, we show that the instance of the CDDG problem has a solution if and only if the answer to the corresponding instance of the 2EV problem is affirmative:

- ($\Rightarrow$) If the cycle detection problem has a solution then the program obtained by taking only the transitions corresponding to the arcs in that cycle satisfies Problem Statement 3.2.1.
- ($\Leftarrow$) Let $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, I_{\mathcal{P}'}\rangle$ be the program obtained after adding two eventually properties. Following the constraints ($C2$) of Problem statement 3.2.1, $I_{\mathcal{P}'} = \{s_u, s_v\}$. Now, consider a computation of $\mathcal{P}'$ that (without loss of generality) starts from s_u. Since $\mathcal{P}'$ satisfies $\Box\Diamond\{s_u\}$, state s_u must be revisited in this computation. Consider the smallest prefix where s_u is repeated. For this prefix, we show the following:

  1. State s_v must occur in this prefix. If not, a computation of $\mathcal{P}'$ that is obtained by repeating the above computation prefix does not satisfy $\Box\Diamond\{s_v\}$.
  2. No other state can be repeated in this computation prefix. If a state, say s_x, appears twice in the above computation prefix then there would be a cycle between the two occurrences of s_x. This implies that, there is a computation of $\mathcal{P}'$ that starts in s_u, reaches s_x and then repeats this cycle. Clearly, this computation does not satisfy $\Box\Diamond\{s_u\}$.

  Now, consider the cycle obtained by taking the edges corresponding to the transitions of the above computation prefix. Based on the first point above, this cycle contains both u and v. And, from the second point, it is a simple cycle, i.e., no vertex is repeated in it. ∎

Corollary 4.2.2 *The problem of adding two or more progress properties to untimed* UNITY *programs is NP-complete.* ∎

4.3 Adding a Single Leads-to Property with Maximum Non-determinism

Given a program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$ and a UNITY specification $SPEC_n$, we say that the revised program $\mathcal{P}'$ has *maximum non-determinism* iff $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, I_{\mathcal{P}'} \rangle$ meets the constraints of Problem Statement 3.2.1 and the cardinality of $T_{\mathcal{P}'}$ is maximum. Maintaining maximum non-determinism is desirable in the sense that it increases the potential for future successful addition of other properties.

Instance. A program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$, $SPEC_n \equiv (P \mapsto Q)$, and positive integer k, where $k \leq |T_{\mathcal{P}}|$.

The decision problem (MND). Given the above instance, does there exist a program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, I_{\mathcal{P}'} \rangle$ such that $\mathcal{P}'$ meets the constraints of Problem Statement 4.1 and $|T_{\mathcal{P}'}| \geq k$?

We now show that the problem of adding a single leads-to property while maintaining maximum non-determinism is NP-complete. To this end, we reduce the *feedback arc set problem* in directed graphs to the above decision problem.

Feedback Arc Set Problem (FAS). Let $G = \langle V, A \rangle$ be a digraph and j be a positive integer, where $j \leq |A|$. The feedback arc set problem determines whether there exists a subset $A' \subseteq A$, such that $|A'| \leq j$ and A' contains at least one arc from every directed cycle in G. The FAS problem is known to be NP-complete [Kar72].

Theorem 4.3.1 *The problem of adding a single leads-to property while preserving maximum non-determinism is NP-complete.*

Proof. Since showing membership to NP is straightforward, we only show that the problem is NP-hard. Given an instance of the FAS problem, we present a polynomial-time mapping from FAS instance to a corresponding instance of the MND problem. Let $G = \langle V, A \rangle$ be a directed graph and j be a positive integer. We construct program $p = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$ and identify integer k and specification $SPEC_n \equiv P \mapsto Q$ as follows:

- $\mathcal{S}$=$\{s_v \mid v \in V\} \,\cup\, \{p_1, p_2 \cdots p_{|A|+1}\} \,\cup\, \{q\}$,
- I=$\{p_1, p_2 \cdots p_{|A|+1}\}$,
- $T_{\mathcal{P}} = \{(s_u, s_v) \mid (u, v) \in A\} \,\cup \{(p_i, s_v) \mid (1 \leq i \leq |A| + 1) \wedge (v \in V)\} \,\cup\, \{(s_v, q) \mid v \in V\} \,\cup\, \{(q, q)\}$,
- $P = \{p_1, p_2 \cdots p_{|A|+1}\}$, and
- $Q = \{q\}$, and $k = |T_{\mathcal{P}}| - j$.

We now show that the instance of FAS has a solution if and only if the answer to the corresponding instance of MND is affirmative:

- ($\Rightarrow$) Let the answer to FAS be the set A' of arcs where $|A'| \leq j$. Clearly, given our mapping, constraints ($C1$) and ($C2$) of the Problem Statement 3.2.1 are met by construction. Now, if we obtain $T_{\mathcal{P}'}$ by removing the transitions that correspond to A' from $T_{\mathcal{P}}$, the resultant program $\mathcal{P}'$ will have no cycles in $\mathcal{S}_{\mathcal{P}} - (P \,\cup\, Q)$. Moreover, since there exists a transition from each state in P to all states in $\mathcal{S}_{\mathcal{P}} - (P \,\cup\, Q)$ and also there exists a transition from each state in $\mathcal{S}_{\mathcal{P}} - P$ to q, any computation that starts from a state in P eventually reaches Q. Observe that the number of transitions removed from $T_{\mathcal{P}}$ is $|A'|$. Hence, $|T_{\mathcal{P}'}| = |T_{\mathcal{P}}| - |A'| \geq |T_{\mathcal{P}}| - j = k$.
- ($\Leftarrow$) Let the answer to MND be the program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, I_{\mathcal{P}'} \rangle$ where $|T_{\mathcal{P}'}| \geq k$. We show that the set $A' = \{(x, y) \mid (s_x, s_v) \in T_{\mathcal{P}} - T_{\mathcal{P}'}\}$ is the answer to FAS. Since $(|A| + 1).|V|$ arcs leave states in P, and, the number of transitions that are removed from $T_{\mathcal{P}}$ (i.e., $|T_{\mathcal{P}} - T_{\mathcal{P}'}|$) is less than $|A|$, any state s_v, where $v \in V$, is reachable from all states in P. Moreover, since $|T_{\mathcal{P}'}| \geq k = |T_{\mathcal{P}}| - j$, it follows that $|A'| = |T_{\mathcal{P}} - T_{\mathcal{P}'}| \leq j$. Now, if there exists a cycle in $\mathcal{P}$, all its transitions must be in the set $\{(s_x, s_y) \mid x, y \in V\}$. Obviously, this cycle is reachable from states in P even though no state in that cycle is in Q. However, this contradicts the assumption that $\mathcal{P}'$ satisfies $P \mapsto Q$. Hence, the set of arcs that correspond to transitions in $T_{\mathcal{P}} - T_{\mathcal{P}'}$ (i.e., $|A'|$) contains at least one arc from each cycle in G. ∎

Chapter 5

Revising Distributed Programs

In this chapter, we shift our focus to *distributed programs* where processes can read and write only a subset of program variables. We expect the concept of program revision to play a more crucial role in the context of distributed programs, since non-determinism and race conditions make it significantly difficult to assert program correctness. We find somewhat unexpected results about the complexity of adding UNITY properties to distributed programs. In particular, we find that the problem of adding only one UNITY safety property or one progress property to a distributed program is NP-complete in the size of the input program's state space.

The knowledge of these complexity bounds is especially important in building tools for incremental synthesis. In particular, the NP-completeness results demonstrate that tools for revising distributed programs must utilize efficient heuristics to expedite the revision algorithm at the cost of *completeness*. Moreover, NP-completeness proofs often identify where the exponential complexity lies in the problem. Thus, thorough analysis of proofs is also crucial in devising efficient heuristics.

With this motivation, in this paper, we also propose an efficient symbolic heuristic that adds a leads-to property to a distributed program. We integrate this heuristic with our tool SYCRAFT (see Chapter 12) that is designed for adding fault-tolerance to existing distributed programs. Meeting leads-to properties are of special interest in fault-tolerant computing where *recovery* within a finite number of steps is essential. To this end, one can first augment the program with all possible recovery transitions that it can use. This augmented program clearly does not guarantee that it would recover to a set of legitimate states, although there is a potential to reach the legitimate states from states reached in the presence of faults. In particular, it may continue to execute on a cycle that is entirely outside the legitimate states. Thus, we apply our heuristic for adding a leads-to property to modify the augmented program so that from any state reachable in the presence of faults, the program is guaranteed recovery to its legitimate states within a finite number of steps. A by-product of the heuristic for adding leads-to properties is a cycle resolution algorithm. Our experimental results show that this algorithm can also be integrated with state-of-the-art model checkers for assisting in developing programs that are correct-by-construction.

This chapter is organized as follows. In Section 5.1, we present our NP-completeness result on revision of distributed programs with respect to UNITY safety properties. In Section 5.2, we show that the problem of adding one progress property to a distributed program is NP-complete. We present our symbolic heuristic for adding a leads-to property to a distributed program and experimental results in Section 5.3.

5.1 Adding UNITY Safety Properties to Distributed Programs

As mentioned in Section 4.1, UNITY safety properties can be characterized by a transition predicate, say $\mathcal{B}$, whose transitions should occur in no computation of a program. In Section 4.1, we also showed that in a centralized setting where processes have no restrictions on reading and writing variables, a program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$ can be easily revised with respect to $\mathcal{B}$ by simply (1) removing the transitions in $\mathcal{B}$ from $T_{\mathcal{P}}$, and (2) making newly created deadlock states unreachable.

To the contrary, the above approach is not adequate for a distributed setting, as it is *sound* (i.e., it constructs a correct program), but not *complete* (i.e., it may fail to find a solution while there exists one). This is due to the issue of read restrictions in distributed programs, which associates each transition of a process with a group predicate. This notion of grouping makes the revision complex, as a revision algorithm has to examine many combinations to determine which group of transitions must be removed and, hence, what deadlock states need to be handled. Indeed, we show that the issue of read restrictions changes the class of complexity of the revision problem entirely.

Instance. A distributed program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$ and a UNITY safety specification $SPEC_n$.

Decision problem. Does there exist a program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, I_{\mathcal{P}'} \rangle$ such that $\mathcal{P}'$ meets the constraints of Problem Statement 3.2.1 for the above instance?

We now show that the above decision problem is NP-complete by a reduction from the well-known *satisfiability* problem. The SAT problem is as follows:

> Let $x_1, x_2 \cdots x_N$ be propositional *variables*. Given a Boolean formula $y = y_{N+1} \wedge y_{N+2} \cdots y_{M+N}$, where each *clause* y_j, $N+1 \leq j \leq M+N$, is a disjunction of three or more literals, does there exist an assignment of truth values to $x_1, x_2 \cdots x_N$ such that y is satisfiable?

We note that the unconventional subscripting of clauses in the above definition of the SAT problem is deliberately chosen to make our proofs simpler.

Theorem 5.1.1 *The problem of adding a* UNITY *safety property to a distributed program is NP-complete.*

Proof. Since showing membership to NP is straightforward, we only need to show that the problem is NP-hard. Towards this end, we present a polynomial-time mapping from an instance of the SAT problem to a corresponding instance of our revision problem. Intuitively, the mapped instance $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$ consists of four processes. Each process is obviously specified by its variables and reachable states using a certain set of transitions. Moreover, each process is subject to certain read/write restrictions. These restrictions along with the assignment of values to variables of processes determine the grouping structure of the instance. The mapped instance also specifies the set of initial states and the safety specification. The structure of the mapped instance is such that the answer to the SAT problem is affirmative if and only if there exists a solution to the revision problem with respect to $\mathcal{P}$. We construct the instance $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$ as follows.

Variables. The set of variables of program $\mathcal{P}$ and, hence, its processes is $V = \{v_0, v_1, v_2, v_3, v_4\}$. The domain of these variables are respectively as follows: $\{-1, 0, 1\}$, $\{-1, 0, 1\}$, $\{0, 1\}$, $\{0, 1\}$, $\{-N \cdots -2, -1, 1, 2 \cdots M+N\} \cup \{j^i \mid (1 \leq i \leq N) \wedge (N+1 \leq$

$j \le M+N)\}$. We note that j^i in the last set is not an exponent, but a denotational symbol.

Reachable states. The set of reachable states in our mapping is as follows:

- For each propositional variable x_i, $1 \le i \le N$, in the instance of the SAT problem, we introduce the following states (see Figure 5.1): $a_i, b_i, b'_i, c_i, c'_i, d_i$, and d'_i. We require that states a_1 and a_{N+1} are identical.
- For each clause y_j, $N+1 \le j \le M+N$, we introduce state r_j.
- For each clause y_j, $N+1 \le j \le M+N$, and variable x_i in clause y_j, $1 \le i \le N$, we introduce the following states: $r_{ji}, s_{ji}, s'_{ji}, t_{ji}$, and t'_{ji}.

Value assignments. Assignment of values to each variable at reachable states is shown in Figure 5.1 (denoted by $< v_0, v_1, v_2, v_3, v_4 >$). We emphasize that assignment of values in our mapping is the most crucial factor in forming group predicates. For reader's convenience, Table 5.1 illustrates the assignment of values to variables more clearly.

(a)

State / Variable name	v_0	v_1	v_2	v_3	v_4
a_i	-1	1	0	1	i
b_i	0	0	0	0	$-i$
b'_i	0	0	0	0	i
c_i	1	0	1	1	$-i$
c'_i	0	1	1	1	i
d_i	0	1	1	1	$-i$
d'_i	1	0	1	1	i

(b)

State / Variable name	v_0	v_1	v_2	v_3	v_4
r_j	0	0	1	0	j
r_{ji}	0	0	0	0	j^i
s_{ji}	0	1	1	1	j^i
s'_{ji}	1	0	1	1	j^i
t_{ji}	1	-1	0	1	j^i
t'_{ji}	-1	-1	0	1	j^i

Table 5.1: Assignment of values to variables in proof of Theorem 5.1.1.

Processes. Program $\mathcal{P}$ consists of four processes. Formally, $\Pi_{\mathcal{P}} = \{p_1, p_2, p_3, p_4\}$. Transition predicate and read/write restrictions of processes in $\Pi_{\mathcal{P}}$ are as follows:

- **Read/write restrictions.** The read/write restrictions of processes p_1, p_2, p_3, and p_4 are as follows:
  - $R_{p_1} = \{v_0, v_2, v_3\}$ and $W_{p_1} = \{v_0, v_2, v_3\}$.
  - $R_{p_2} = \{v_1, v_2, v_3\}$ and $W_{p_2} = \{v_1, v_2, v_3\}$.

- $R_{p_3} = \{v_0, v_1, v_2, v_3, v_4\}$ and $W_{p_3} = \{v_0, v_1, v_2, v_4\}$.
- $R_{p_4} = \{v_0, v_1, v_2, v_3, v_4\}$ and $W_{p_4} = \{v_0, v_1, v_3, v_4\}$.

- **Transition predicates.** For each propositional variable x_i, $1 \leq i \leq N$, we include the following transitions in processes p_1, p_2, p_3, and p_4 (see Figure 5.1):

  - $T_{p_1} = \{(b'_i, d'_i), (b_i, c_i) \mid 1 \leq i \leq N\}$.
  - $T_{p_2} = \{(b'_i, c'_i), (b_i, d_i) \mid 1 \leq i \leq N\}$.
  - $T_{p_3} = \{(c'_i, a_{i+1}), (c_i, a_{i+1}), (d'_i, a_{i+1}), (d_i, a_{i+1}) \mid 1 \leq i \leq N\}$.
  - $T_{p_4} = \{(a_i, b_i), (a_i, b'_i) \mid 1 \leq i \leq N\}$.

  Moreover, corresponding to each clause y_j, $N+1 \leq j \leq M+N$, and variable x_i, $1 \leq i \leq N$, in clause y_j, we include transition (r_j, r_{ji}) in T_{p_3} and the following:

  - If x_i is a literal in clause y_j, then we include transition (r_{ji}, s_{ji}) in T_{p_2}, (s_{ji}, t_{ji}) in T_{p_3}, and (t_{ji}, b_i) in T_{p_4}.
  - If $\neg x_i$ is a literal in clause y_j, then we include transition (r_{ji}, s'_{ji}) in T_{p_1}, (s'_{ji}, t'_{ji}) in T_{p_3}, and (t'_{ji}, b'_i) in T_{p_4}.

  Note that only for the sake of illustration, Figure 5.1 shows all possible transitions. However, in order to construct $\mathcal{P}$, based on the existence of x_i or $\neg x_i$ in y_j, we only include a subset of the transitions.

Initial states. The set $I_{\mathcal{P}}$ of initial states represents clauses of the instance of the SAT problem, i.e., $I_{\mathcal{P}} = \{r_j \mid N+1 \leq j \leq M+N\}$.

Safety property. Let P be a state predicate that contains all reachable states in Figure 5.1 except c_i and c'_i (i.e., $c_i, c'_i \in \neg P$). Thus, the properties $\mathsf{stable}\ P$ and $\mathsf{invariant}\ P$ can be characterized by the transition predicate $\mathcal{B} = \{(b_i, c_i), (b'_i, c'_i) \mid 1 \leq i \leq N\}$. Similarly, let P and Q be two state predicates that contain all reachable states in Figure 5.1 except c_i and c'_i. Thus, the safety property $P\ \mathsf{unless}\ Q$ can be characterized by $\mathcal{B}$ as well. In our mapping, we let $\mathcal{B}$ represent the safety specification for which $\mathcal{P}$ has to be revised.

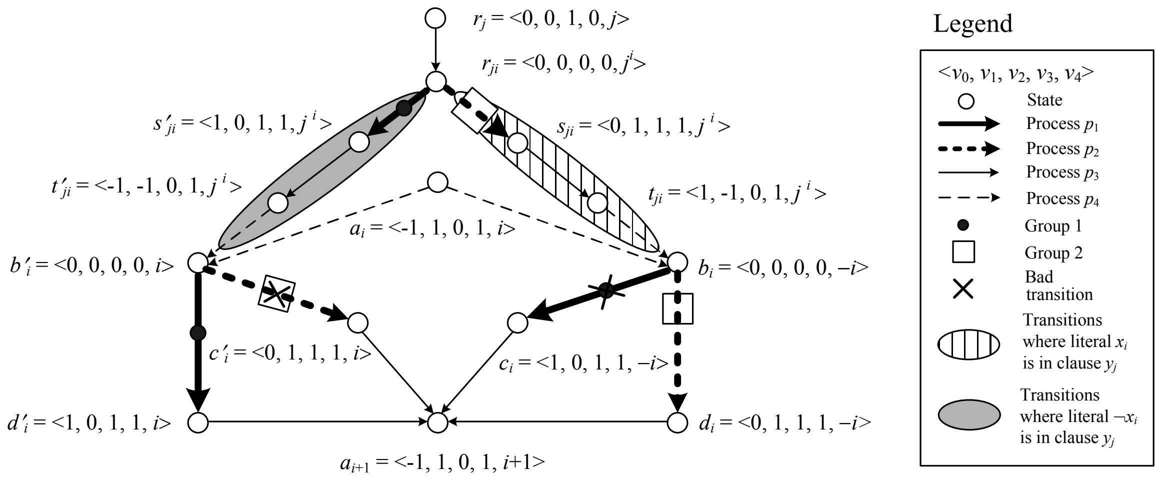


Figure 5.1: Mapping SAT to addition of UNITY safety properties.

Before we present our reduction from the SAT problem using the above mapping, we make the following observations regarding the grouping of transitions in different processes:

1. Due to inability of process p_1 to read variable v_4, for all i, $1 \leq i \leq N$, transitions $(r_{ji}, s'_{ji}), (b'_i, d'_i)$, and (b_i, c_i) are grouped in p_1.

2. Due to inability of process p_2 to read variable v_4, for all i, $1 \leq i \leq N$, transitions $(r_{ji}, s_{ji}), (b_i, d_i)$, and (b'_i, c'_i) are grouped in p_2.

3. Transitions grouped with the rest of the transitions in Figure 5.1 are unreachable and, hence, are irrelevant.

Now, we show that the answer to the SAT problem is affirmative if and only if there exists a solution to the revision problem. Thus, we distinguish two cases:

- $(\Rightarrow)$ First, we show that if the given instance of the SAT formula is satisfiable, then there exists a solution that meets the requirements of the revision decision problem. Since the SAT formula is satisfiable, there exists an assignment of truth values to all variables x_i, $1 \leq i \leq N$, such that each y_j, $N+1 \leq j \leq M+N$, is true. Now, we identify a program $\mathcal{P}'$, that is obtained by adding the safety property represented by $\mathcal{B}$ to program $\mathcal{P}$ as follows.

  - The state space of $\mathcal{P}'$ consists of all the states of $\mathcal{P}$, i.e., $\mathcal{S}_{\mathcal{P}} = \mathcal{S}_{\mathcal{P}'}$.
  - The initial states of $\mathcal{P}'$ consists of all the initial states of $\mathcal{P}$, i.e., $I_{\mathcal{P}} = I_{\mathcal{P}'}$.

– For each variable x_i, $1 \leq i \leq N$, if x_i is *true*, then we include the following transitions: (a_i, b_i) in T_{p_4}, (b_i, d_i) in T_{p_2}, and (d_i, a_{i+1}) in T_{p_3}.

– For each variable x_i, $1 \leq i \leq N$, if x_i is *false*, then we include the following transitions:(a_i, b'_i) in T_{p_4}, (b'_i, d'_i) in T_{p_1}, and (d'_i, a_{i+1}) in T_{p_3}.

– For each clause y_j, $N+1 \leq j \leq M+N$, that contains literal x_i, if x_i is *true*, we include the following transitions: (r_j, r_{ji}) and (s_{ji}, t_{ji}) in T_{p_3}, (r_{ji}, s_{ji}) in T_{p_2}, and (t_{ji}, b_i) in T_{p_4}.

– For each clause y_j, $N+1 \leq j \leq M+N$, that contains literal $\neg x_i$, if x_i is *false*, we include the following transitions: (r_j, r_{ji}) and (s'_{ji}, t'_{ji}) in T_{p_3}, (r_{ji}, s'_{ji}) in T_{p_1}, and (t'_{ji}, b'_i) in T_{p_4}.

As an illustration, we show the partial structure of $\mathcal{P}'$, for the formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_4)$, where $x_1 = true$, $x_2 = false$, $x_3 = false$, and $x_4 = false$, in Figure 5.2. Notice that states whose all outgoing and incoming transitions are eliminated are not illustrated. Now, we show that $\mathcal{P}'$ meets the requirements of the Problem Statement 3.2.1:

1. The first three constraints of the decision problem are trivially satisfied by construction.

2. We now show that constraint $C4$ holds. First, it is easy to observe that by construction, there exist no reachable deadlock states in the revised program. Hence, if $\mathcal{P}$ refines UNITY specification $SPEC_e$, then $\mathcal{P}'$ refines $SPEC_e$ as well. Moreover, if a computation of $\mathcal{P}'$ reaches a state b_i for some i, from an initial state r_j (i.e., x_i is *true* in clause y_j), then that computation cannot violate safety since bad transition (b_i, c_i) is removed. This is due to the fact that (b_i, c_i) is grouped with transition (r_{ji}, s'_{ji}) and this transition is not included in $T_{\mathcal{P}'}$, as literal x_i is *true* in y_j. Likewise, if a computation of $\mathcal{P}'$ reaches a state b'_i for some i, from initial state r_j (i.e., x_i is *false* in clause y_j), then that computation cannot violate safety since transition (b'_i, c'_i) is removed. This is due to the fact that (b'_i, c'_i) is grouped with transition (r_{ji}, s_{ji}) and this transition is not included in $T_{\mathcal{P}'}$, as x_i is *false*. Thus, $\mathcal{P}'$ refines $SPEC_n$.

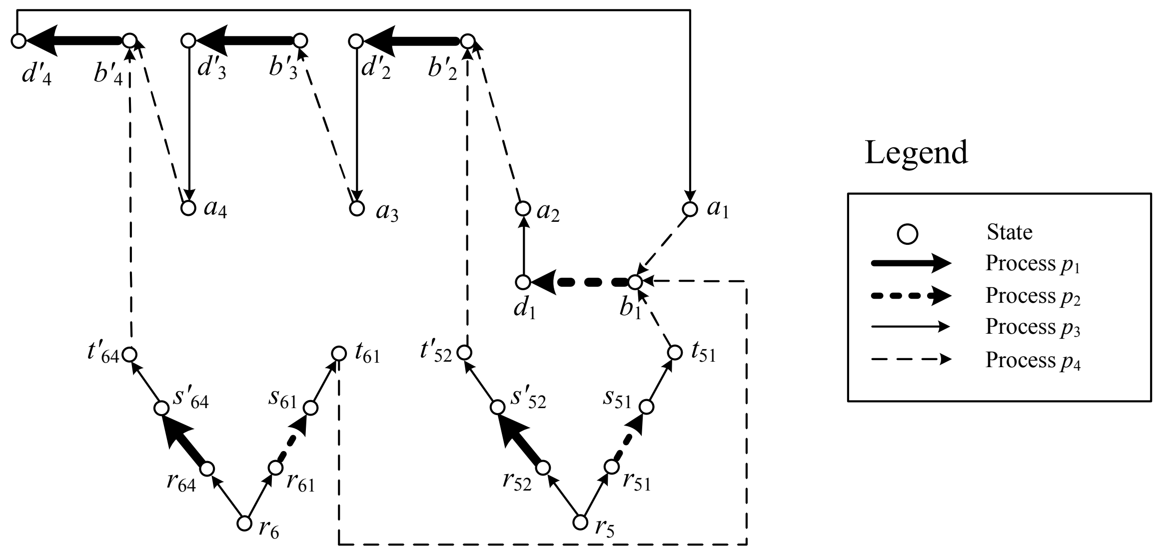


Figure 5.2: The structure of the revised program for Boolean formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_4)$, where $x_1 = \mathit{true}$, $x_2 = \mathit{false}$, $x_3 = \mathit{false}$, and $x_4 = \mathit{false}$.

- ($\Leftarrow$) Next, we show that if there exists a solution to the revision problem for the instance identified by our mapping from the SAT problem, then the given SAT formula is satisfiable. Let $\mathcal{P}'$ be the program that is obtained by adding the safety property $SPEC_n$ to program $\mathcal{P}$. Now, in order to obtain a solution for SAT, we proceed as follows. If there exists a computation of $\mathcal{P}'$ where state b_i is reachable, then we assign x_i the truth value *true*. Otherwise, we assign the truth value *false*.

  We now show that the above truth assignment satisfies all clauses. Let y_j be a clause for some j, $N+1 \leq j \leq M+N$, and let r_j be the corresponding initial state in $I_{\mathcal{P}'}$. Since r_j is an initial state and $\mathcal{P}'$ cannot deadlock, the transition (r_j, r_{ji}) must be present in $T_{\mathcal{P}'}$, for some i, $1 \leq i \leq N$. By the same argument, there must exist some transition that originates from r_{ji}. This transition terminates in either s_{ji} or s'_{ji}. Observe that $T_{\mathcal{P}'}$ cannot have both transitions, as grouping of transitions will include both (b_i, c_i) and (b'_i, c'_i) which in turn causes violation of safety by $\mathcal{P}'$. Now, if the transition from r_{ji} terminates in s_{ji}, then clause y_j contains literal x_i and x_i is assigned the truth value *true*. Hence, y_j evaluates to true. Likewise, if the transition from r_{ji} terminates in s'_{ji}, then clause y_j contains literal $\neg x_i$ and x_i is assigned the truth value *false*. Hence, y_j evaluates to true. Therefore, the assignment of values considered above is a satisfying truth assignment for the given SAT formula. ∎

5.2 Adding UNITY Progress Properties to Distributed Programs

In a centralized setting, where programs have no restriction on reading and writing variables, a program, say $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$, can be easily revised with respect to a progress property by simply (1) breaking non-progress cycles that prevent a program to eventually reach a desirable state predicate, and (2) removing deadlock states (see Section 4.1). To the contrary, in a distributed setting, due to the issue of grouping, it matters which transition (and as a result its corresponding group) is removed to break a non-progress cycle.

Instance. A distributed program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$ and a Unity progress property $SPEC_n$.

Decision problem. Does there exist a program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, I_{\mathcal{P}'} \rangle$ such that $\mathcal{P}'$ meets the constraints of Problem Statement 3.2.1 for the above instance?

Theorem 5.2.1 *The problem of adding a* Unity *progress property to a distributed program is NP-complete.*

Proof. Since showing membership to NP is straightforward, we only show that the problem is NP-hard by a reduction from the SAT problem. First, we present a polynomial-time mapping.

Variables. The set of variables of program $\mathcal{P}$ and, hence, its processes is $V = \{v_0, v_1, v_2, v_3, v_4\}$. The domain of these variables are respectively as follows: $\{0,1\}$, $\{0,1\}$, $\{-N \cdots -2, -1, 1, 2 \cdots M+N\} \cup \{j^i \mid (1 \leq i \leq N) \wedge (N+1 \leq j \leq M+N)\}$, $\{-1,0,1\}$, and $\{-1,0,1\}$.

Reachable states. The set of reachable states in our mapping is as follows:

- For each propositional variable x_i, $1 \leq i \leq N$, we introduce the following states (see Figure 5.3): a_i, a'_i, b_i, b'_i, c_i, c'_i, d_i, d'_i, Q_i, and Q'_i.
- For each clause y_j, $N+1 \leq j \leq M+N$, we introduce state r_j.

- For each clause y_j, $N+1 \leq j \leq M+N$, and variable x_i, $1 \leq i \leq N$, in clause y_j, we introduce states r_{ji}, s_{ji}, and s'_{ji}.

Value assignments. Assignment of values to each variable at reachable states is shown in Figure 5.3 (denoted by $< v_0, v_1, v_2, v_3, v_4 >$). For reader's convenience, Table 5.2 illustrates the assignment of values to variables more clearly.

(a)

State / Variable name	v_0	v_1	v_2	v_3	v_4
a_i	1	0	$-i$	-1	-1
a'_i	1	0	i	-1	1
b_i	0	0	$-i$	0	0
b'_i	0	0	i	0	0
c_i	1	1	$-i$	0	1
c'_i	1	1	i	1	0
d_i	0	1	i	1	-1
d'_i	0	1	$-i$	1	1
Q_i	1	1	$-i$	1	0
Q'_i	1	1	i	0	1

(b)

State / Variable name	v_0	v_1	v_2	v_3	v_4
r_j	0	1	j	1	1
r_{ji}	0	0	j^i	0	0
s_{ji}	1	1	j^i	0	1
s'_{ji}	1	1	j^i	1	0

Table 5.2: Assignment of values to variables in proof of Theorem 5.2.1.

Processes. Program $\mathcal{P}$ consists of four processes. Formally, $\Pi_{\mathcal{P}} = \{p_1, p_2, p_3, p_4\}$. Transition predicate and read/write restrictions of processes in $\Pi_{\mathcal{P}}$ are as follows:

- **Read/write restrictions.** The read/write restrictions of processes p_1, p_2, p_3, and p_4 are as follows:
  - $R_{p_1} = \{v_0, v_1, v_3\}$ and $W_{p_1} = \{v_0, v_1, v_3\}$.
  - $R_{p_2} = \{v_0, v_1, v_4\}$ and $W_{p_2} = \{v_0, v_1, v_4\}$.
  - $R_{p_3} = \{v_0, v_1, v_2, v_3, v_4\}$ and $W_{p_3} = \{v_0, v_2, v_3, v_4\}$.
  - $R_{p_4} = \{v_0, v_1, v_2, v_3, v_4\}$ and $W_{p_4} = \{v_1, v_2, v_3, v_4\}$.

- **Transition predicates.** For each propositional variable x_i, $1 \leq i \leq N$, we include the following transitions in processes p_1, p_2, p_3, and p_4 (see Figure 5.3):
  - $T_{p_1} = \{(b'_i, c'_i), (b_i, Q_i) \mid 1 \leq i \leq N\}$.
  - $T_{p_2} = \{(b_i, c_i), (b'_i, Q'_i) \mid 1 \leq i \leq N\}$.

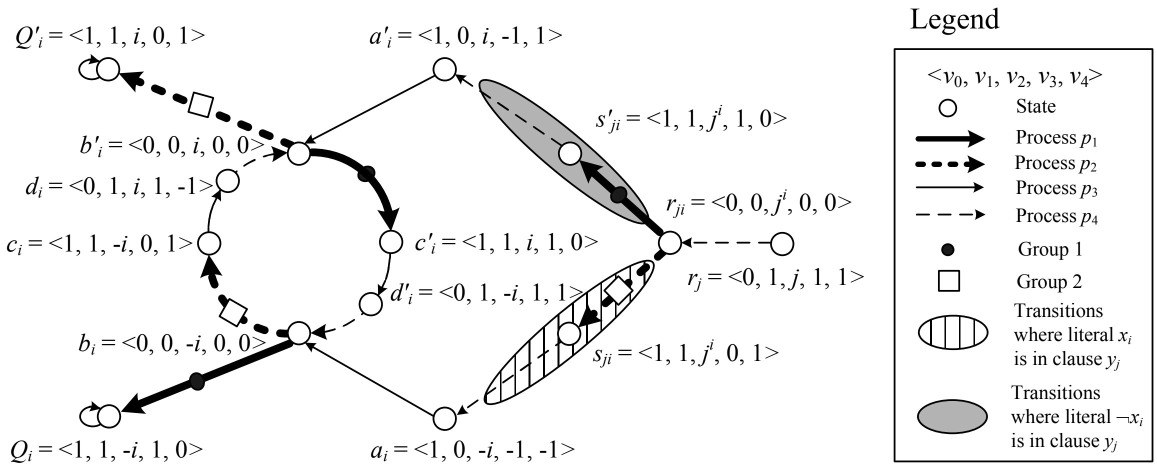


Figure 5.3: Mapping SAT to addition of a progress property.

- $T_{p_3} = \{(a_i, b_i), (a'_i, b'_i), (c_i, d_i), (c'_i, d'_i), (Q_i, Q_i), (Q'_i, Q'_i) \mid 1 \leq i \leq N\}$.
- $T_{p_4} = \{(d'_i, b_i), (d_i, b'_i) \mid 1 \leq i \leq N\}$.

Moreover, corresponding to each clause y_j, $N+1 \leq j \leq M+N$, and variable x_i, $1 \leq i \leq N$, in clause y_j, we include transition (r_j, r_{ji}) in T_{p_4} and the following:

- If x_i is a literal in clause y_j, then we include transition (r_{ji}, s_{ji}) in T_{p_2}, and (s_{ji}, a_i) in T_{p_4}.
- If $\neg x_i$ is a literal in clause y_j, then we include transition (r_{ji}, s'_{ji}) in T_{p_1} and (s'_{ji}, a'_i) in T_{p_4}.

Note that for the sake of illustration, Figure 5.3 shows all possible transitions. However, in order to construct $\mathcal{P}'$, based on the existence of x_i or $\neg x_i$ in y_j, we only include a subset of transitions.

Initial states. The set $I_{\mathcal{P}}$ of initial states represents clauses of the Boolean formula in the instance of the SAT problem, i.e., $I_{\mathcal{P}} = \{r_j \mid N+1 \leq j \leq M+N\}$.

Progress property. In our mapping, the desirable progress property is of the form $SPEC_n \equiv (true \text{ leads-to } Q)$, where $Q = \{Q_i, Q'_i \mid 1 \leq i \leq N\}$ (see Figure 5.3). Observe that $SPEC_n$ is a leads-to as well as an ensures property. This property in Linear Temporal Logic (LTL) is denoted by $\Box\Diamond Q$ (called *always eventually* Q).

Before we present our reduction from the SAT problem using the above mapping, we make the following observations regarding the grouping of transitions in different processes:

1. Due to inability of process p_1 to read variable v_2, for all i, $1 \leq i \leq N$, transitions (r_{ji}, s'_{ji}), (b'_i, c'_i), and (b_i, Q_i) are grouped in process p_1.

2. Due to inability of process p_2 to read variable v_2, for all i, $1 \leq i \leq N$, transitions (r_{ji}, s_{ji}), (b_i, c_i), and (b'_i, Q'_i) are grouped in process p_2.

3. Transitions grouped with the rest of the transitions in Figure 5.3 are unreachable and, hence, are irrelevant.

We distinguish the following two cases for reducing the SAT problem to our revision problem :

- ($\Rightarrow$) First, we show that if the given instance of the SAT formula is satisfiable, then there exists a solution that meets the requirements of the revision decision problem. Since the SAT formula is satisfiable, there exists an assignment of truth values to all variables x_i, $1 \leq i \leq N$, such that each y_j, $N+1 \leq j \leq M+N$, is true. Now, we identify a program $\mathcal{P}'$, that is obtained by adding the progress property $\Box\Diamond Q$ to program $\mathcal{P}$ as follows.

  - The state space of $\mathcal{P}'$ consists of all the states of $\mathcal{P}$, i.e., $\mathcal{S}_{\mathcal{P}} = \mathcal{S}_{\mathcal{P}'}$.
  - The initial states of $\mathcal{P}'$ consists of all the initial states of $\mathcal{P}$, i.e., $I_{\mathcal{P}} = I_{\mathcal{P}'}$.
  - For each variable x_i, $1 \leq i \leq N$, if x_i is *true*, then we include the following transitions: (a_i, b_i), (c_i, d_i), and (Q'_i, Q'_i) in T_{p_3}, (b_i, c_i) and (b'_i, Q'_i) in T_{p_2}, and, (d_i, b'_i) in T_{p_4}.
  - For each variable x_i, $1 \leq i \leq N$, if x_i is *false*, then we include the following transitions: (a'_i, b'_i), (c'_i, d'_i), and (Q_i, Q_i) in T_{p_3}, (b'_i, c'_i) and (b_i, Q_i) in T_{p_1}, and, (d'_i, b_i) in T_{p_4}.
  - For each clause y_j, $N+1 \leq j \leq M+N$, that contains literal x_i, if x_i is *true*, we include transitions (r_j, r_{ji}) and (s_{ji}, a_i) in T_{p_4}, and, transition (r_{ji}, s_{ji}) in T_{p_2}.
  - For each clause y_j, $N+1 \leq j \leq M+N$, that contains literal $\neg x_i$, if x_i is *false*, we include transitions (r_j, r_{ji}) and (s'_{ji}, a'_i) in T_{p_4}, and, transition (r_{ji}, s'_{ji}) in T_{p_1}.

As an illustration, we show the partial structure of $\mathcal{P}'$, for the formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_4)$, where $x_1 = \mathit{true}$, $x_2 = \mathit{false}$, $x_3 = \mathit{false}$, and $x_4 = \mathit{false}$ in Figure 5.4. Notice that states whose all outgoing and incoming transitions are eliminated are not illustrated. Now, we show that $\mathcal{P}'$ meets the requirements of the Problems Statement 3.2.1:

1. The first three constraints of the decision problem are trivially satisfied by construction.
2. We now show that constraint $C4$ holds. First, it is easy to observe that by construction, there exist no reachable deadlock states in the revised program. Hence, if $\mathcal{P}$ refines UNITY specification SPEC_e, then $\mathcal{P}'$ refines SPEC_e as well. Moreover, by construction, all computations of $\mathcal{P}'$ eventually reach either Q_i or Q'_i and will stutter there. This is due to the fact that if literal x_i is *true* in clause y_j, then transition (r_{ji}, s'_{ji}) is not included in $T_{\mathcal{P}'}$ and, hence, its group-mates (b'_i, c'_i) and (b_i, Q_i) are not in $T_{\mathcal{P}'}$ as well. Consequently, a computation that starts from r_j eventually reaches Q'_i without meeting a cycle. Likewise, if literal $\neg x_i$ is *false* in clause y_j, then transition (r_{ji}, s_{ji}) is not included in $T_{\mathcal{P}'}$ and, hence, its group-mates (b_i, c_i) and (b'_i, Q'_i) are not in $T_{\mathcal{P}'}$ as well. Consequently, a computation that starts from r_j eventually reaches Q_i without meeting a cycle. Hence, $\mathcal{P}'$ refines $\mathit{SPEC}_n \equiv \Box\Diamond Q$.

- ($\Leftarrow$) Next, we show that if there exists a solution to the revision problem for the instance identified by our mapping from the SAT problem, then the given SAT formula is satisfiable. Let $\mathcal{P}'$ be the program that is obtained by adding the progress property in $\mathit{SPEC}_n \equiv \Box\Diamond Q$ to program $\mathcal{P}$. Now, in order to obtain a solution for SAT, we proceed as follows. If there exists a computation of $\mathcal{P}'$ where state a_i is reachable, then we assign x_i the truth value *true*. Otherwise, we assign the truth value *false*.

  We now show that the above truth assignment satisfies all clauses. Let y_j be a clause for some j, $N+1 \leq j \leq M+N$, and let r_j be the corresponding initial state in $I_{\mathcal{P}'}$. Since r_j is an initial state and $\mathcal{P}'$ cannot deadlock, the transition (r_j, r_{ji}) must be present in $T_{\mathcal{P}'}$, for some i, $1 \leq i \leq N$. By the same argument, there must exist some transition that originates from r_{ji}. This transition terminates in either s_{ji} or s'_{ji}.

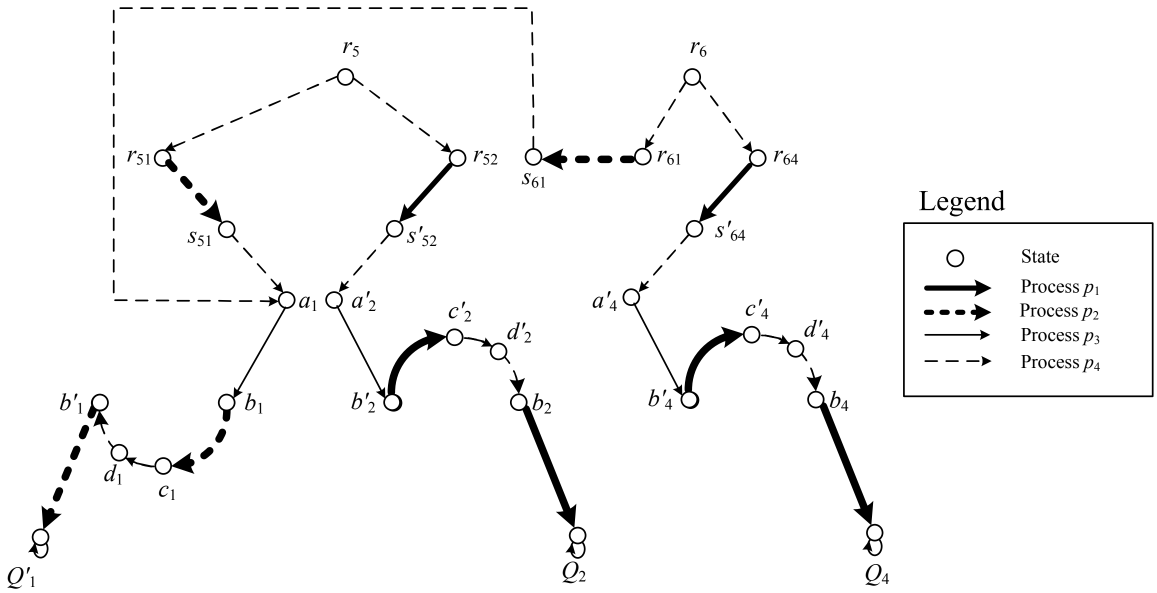


Figure 5.4: The structure of the revised program for Boolean formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_4)$, where $x_1 = \mathit{true}$, $x_2 = \mathit{false}$, $x_3 = \mathit{false}$, and $x_4 = \mathit{false}$.

Observe that $T_{\mathcal{P}'}$ cannot have both transitions, as grouping of transitions will include transitions (b_i, c_i) and (b'_i, c'_i). If this is the case, $\mathcal{P}'$ does not refine the property $\Box\Diamond Q$ due to the existence of cycle $b_i \rightarrow c_i \rightarrow d_i \rightarrow b'_i \rightarrow c'_i \rightarrow d'_i \rightarrow b_i$. Thus, there can be one and only one outgoing transition from r_{ji} in $T_{\mathcal{P}'}$. Now, if the transition from r_{ji} terminates in s_{ji}, then clause y_j contains literal x_i and x_i is assigned the truth value *true*. Hence, y_j evaluates to true. Likewise, if the transition from r_{ji} terminates in s'_{ji}, then clause y_j contains literal $\neg x_i$ and x_i is assigned the truth value *false*. Hence, y_j evaluates to true. Therefore, the assignment of values considered above is a satisfying truth assignment for the given SAT formula. ∎

5.3 A Symbolic Heuristic for Adding Leads-To Properties

We now present a polynomial-time (in the size of the state space) symbolic (BDD[1]-based) heuristic for adding leads-to properties to distributed programs. Leads-to properties have interesting applications in automated addition of recovery for synthesizing fault-tolerant distributed programs.

[1]Ordered Binary Decision Diagrams [Bry86] represent Boolean formulae as directed acyclic graphs making testing of functional properties such as satisfiability and equivalence straightforward and extremely efficient.

Algorithm 5.1 Add_LeadsTo

Input: A distributed program $\mathcal{P} = \langle \Pi_\mathcal{P}, I_\mathcal{P} \rangle$ and property P leads-to Q.
Output: If successful, transition predicate $T_{\mathcal{P}'}$ of the new program.

1:
2: **repeat**
3: Let Rank[i] be the state predicate whose length of shortest path to Q is i, where Rank[0] $= Q$ and Rank[∞] = the state predicate that is reachable from P, but cannot reach Q;
4: **for all** i and j **do**
5: C := ComputeCycles($T_\mathcal{P}, P$);
6: **if** $(i \leq j) \ \wedge \ (i \neq 0) \ \wedge \ (i \neq \infty)$ **then**
7: tmp := $Group(\langle C \wedge \text{Rank}[i] \rangle \ \wedge \ \langle C \wedge \text{Rank}[j] \rangle')$;
8: **if** removal of tmp from $T_\mathcal{P}$ eliminates a state from Q **then**
9: Make $\langle C \wedge tmp \rangle$ unreachable;
10: **else**
11: $T_\mathcal{P} \ := \ T_\mathcal{P} - tmp$;
12: **end if**
13: **end if**
14: **end for**
15: **until** Rank[∞] $= \{\}$
16: $T_{\mathcal{P}'}$:= EliminateDeadlockStates(P, Q, $\langle \Pi_\mathcal{P}, I_\mathcal{P} \rangle$);
17: **return** $T_{\mathcal{P}'}$;

The NP-hardness reduction presented in the proof of Theorem 5.2.1 precisely shows where the complexity of the problem lies in. Indeed, Figure 5.3 shows that transition (b_i, c_i) which can potentially be removed to break the non-progress cycle $b_i \rightarrow c_i \rightarrow d_i \rightarrow b'_i \rightarrow c'_i \rightarrow d'_i \rightarrow b_i$ is grouped with the critical transition (r_{ji}, s_{ji}) which ensures that state r_{ji} and consequently initial state r_j are not deadlocked. The same argument holds for transitions (b'_i, c'_i) and (r_{ji}, s'_{ji}). Thus, a heuristic that adds a leads-to property to a distributed program needs to address this issue.

Our heuristic works as follows (cf. Algorithm 5.1). The Algorithm Add_LeadsTo takes a distributed program $\mathcal{P} = \langle \Pi_\mathcal{P}, I_\mathcal{P} \rangle$ and a property P leads-to Q as input, where P and Q are two arbitrary state predicates in the state space of $\mathcal{P}$. The algorithm (if successful) returns transition predicate of the derived program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, I_{\mathcal{P}'} \rangle$ that refines P leads-to Q as output. In order to transform $\mathcal{P}$ to $\mathcal{P}'$, first, the algorithm ranks states that can be reached from P based on the length of their shortest path to Q (Line 2). Then, it attempts to break non-progress cycles (Lines 3-13). To this end, it first computes the set of cycles that are reachable from P (Line 4). This computation can be accomplished using any BDD-based cycle detection algorithm. We apply the Emerson-Lie method [EL86]. Then, the algorithm removes transitions from $T_{\mathcal{P}}$ that participate in a cycle and whose rank of source state is less than or equal to the rank of destination state (Lines 6-10). However, since removal of a transition must take place with its entire group predicate, we do not remove a transition that causes creation of deadlock states in Q. Instead, we make the corresponding cycle unreachable (Line 8). This can be done by simply removing transitions that terminate in a state on the cycle. Thus, if removal of a group of transitions does not create new deadlock states in Q, the algorithm removes them (Line 10). Finally, since removal of transitions may create deadlock states outside Q but reachable from P, we need to eliminate those deadlock states (Line 15). Such elimination can be accomplished using the BDD-based method proposed in [BK07b].

Given $O(n^2)$ complexity of the cycle detection algorithm [EL86], it is straightforward to observe that the complexity of our heuristic is $O(n^4)$, where n is the size of state space of $\mathcal{P}$. In order to evaluate the performance of our heuristic, we have implemented the Algorithm Add_LeadsTo in our tool SYCRAFT [BK08c]. This heuristic can be used for adding *recovery* in order to synthesize fault-tolerant distributed programs as follows. Let S be a set of legitimate states (e.g., an invariant predicate) and T be the *fault-span* predicate (i.e., the set of states reachable in the presence of faults). First, we add all possible transitions that start from $T - S$ and end in T. Then, we apply the Algorithm Add_LeadsTo for property $(T - S)$ leads-to S.

Table 5.3 illustrates experimental results of our heuristic for adding such recovery. All experiments are run on a PC with a 2.8GHz Intel Xeon processor and 1.2GB RAM. The BDD representation of the Boolean formulae has been done using the Glu/CUDD package [Som]. Our experiments target addition of recovery to two well-known problems in fault-tolerant distributed computing, namely, the *Byzantine agreement* problem [LSP82] (denote $\mathcal{BA}^i$) and the *token ring* problem [AK98a] (denoted $\mathcal{TR}^i$), where i is the number of processes. Table 5.3 shows the size of reachable states in the presence of faults, memory usage, total time spent to add the desirable leads-to property, time spent for cycle detection (i.e., Line 4 in Algorithm 5.1), and time spent for breaking cycles by pruning transitions. Given the huge size of reachable states and complexity of structure of programs in our experiments, we find the experimental results quite encouraging. We note that the reason that $\mathcal{TR}$ and $\mathcal{BA}$ behave differently as their number of processes grow is due to their different

	Space		*Time(s)*		
	reachable states	**memory (KB)**	**cycle detection**	**pruning transitions**	**total**
$\mathcal{BA}^5$	10^4	12	0.5	2.5	3
$\mathcal{BA}^{10}$	10^8	18	5	18	23
$\mathcal{BA}^{15}$	10^{12}	26	47	76	125
$\mathcal{BA}^{20}$	10^{16}	29	522	372	894
$\mathcal{BA}^{25}$	10^{20}	30	3722	1131	4853
$\mathcal{TR}^5$	10^2	6	0.2	0.3	0.5
$\mathcal{TR}^{10}$	10^5	7	13	2	15
$\mathcal{TR}^{15}$	10^7	10	470	10	480
$\mathcal{TR}^{20}$	10^9	33	2743	173	2916
$\mathcal{TR}^{25}$	10^{11}	53	22107	2275	24382

Table 5.3: Experimental results of the symbolic heuristic.

structures, existing cycles, and number of reachable states. In particular, the state space of $\mathcal{TR}$ is highly reachable and its original program has a cycle that includes all of its legitimate states. This is not the case in $\mathcal{BA}$. We also note that in case of $\mathcal{TR}$, the symbolic heuristic presented in this subsection tend to be slower than the constructive layered approach introduced in [BK07b]. However, the approach in this paper is more general and has a better potential of success than the approach in [BK07b].

Chapter 6

Revising Real-Time Programs

In this chapter, we focus on automated revision of *real-time* Unity programs. Our focus in this chapter is mainly on addition of bounded-time leads-to properties to real-time programs. This is due to the following two reasons:

1. the complexity of addition of other real-time Unity properties (i.e., bounded-time unless, bounded-time stable, and bounded-time ensures) can be easily shown to be in the same class as bounded-time leads-to, and

2. bounded-time leads-to is typically used in specifying most of the interesting properties of real-time systems such as meeting a deadline and converging to normal behavior of a system.

The rest of this chapter is organized as follows. In Section 6.1, we present a sound and complete algorithm for adding a bounded-time leads-to property to a real-time program. The complexity of this algorithm is in polynomial-time in the size of the input program's region graph. Then, in Section 6.2, we show that the problem of adding a bounded-time leads-to property to a real-time program while maintaining maximum non-determinism is NP-complete in the size of the program's region graph even if the given program satisfies the corresponding unbounded leads-to property. Finally, in Section 6.3, we show that the problem of adding an interval bounded-time leads-to property to a real-time program is also NP-complete in the size of the input program's region graph.

6.1 Adding a Single Bounded-Time **Leads-to** Property

In this subsection, we present a sound and complete algorithm that automatically adds a single bounded-time *leads-to* property to a real-time program.

Algorithm sketch. Intuitively, the algorithm works in four phases. In Phase 1, we transform the input real-time program into a region graph and subsequently a weighted directed graph (called MaxDelay digraph [CY91]). The property of this digraph is such that the longest distance between any two vertices is equal to the maximum time delay between the corresponding regions in the region graph. Then, in Phase 2, we identify a subgraph of the MaxDelay digraph in which the desired bounded-time leads-to property is never violated. In Phase 3, we remove deadlock regions. Finally, in Phase 4, we transform the resultant region graph back into a real-time program.

Construction of MaxDelay digraph. We now describe how we transform a region graph $R(\mathcal{P}) = \langle \Pi^r_{\mathcal{P}}, I^r_{\mathcal{P}} \rangle$ into a MaxDelay digraph $G = \langle V, A \rangle$. Vertices of G consist of the regions in $R(\mathcal{P})$.

Notation: We denote the weight of an arc $(v_0, v_1) \in A$ by $Weight(v_0, v_1)$. Let $\gamma : \mathcal{S}^r_{\mathcal{P}} \leftrightarrow V$ denote a bijection that maps each region $r \in \mathcal{S}^r_{\mathcal{P}}$ to its corresponding vertex in G and vice versa, i.e., $\gamma(r)$ is a vertex of G that represents region r of $R(\mathcal{P})$ and $\gamma^{-1}(v)$ is the region of $R(\mathcal{P})$ that corresponds to vertex v in V. Let $\Gamma : 2^{\mathcal{S}^r_{\mathcal{P}}} \leftrightarrow 2^V$ be a bijection that maps a region predicate in $R(\mathcal{P})$ to the corresponding set of vertices of G and let Γ^{-1} be its inverse. Finally, for a boundary region r with respect to clock variable x, we denote the value of x by $r.x$ (equal to some nonnegative integer).

Arcs of G consist of the following:

- Arcs of weight 0 from v_0 to v_1, if $\gamma^{-1}(v_0) \to \gamma^{-1}(v_1)$ represents an immediate transition in $R(\mathcal{P})$.

- Arcs of weight $c' - c$ from v_0 to v_1, where $c, c' \in \mathbb{Z}_{\geq 0}$ and $c' > c$, if $\gamma^{-1}(v_0)$ and $\gamma^{-1}(v_1)$ are both boundary regions with respect to clock variable x, such that $\gamma^{-1}(v_0).x = c$, $\gamma^{-1}(v_1).x = c'$, and there is a path in $R(\mathcal{P})$ from $\gamma^{-1}(v_0)$ to $\gamma^{-1}(v_1)$ which does not reset x.

Algorithm 6.1 Add_rtUNITY

Input: real-time program $\langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$ and bounded leads-to property $P \mapsto_{\tau} Q$.
Output: revised program $\langle \Pi_{\mathcal{P}'}, I_{\mathcal{P}'} \rangle$.

Let $c_t := \tau$ where t is a new clock variable; {Phase 1}
$\forall ((l_0, \nu) \to (l_1, \nu[\lambda := 0])) \in \delta_p \mid (l_0 \notin P \wedge l_1 \in P) : \lambda := \lambda \cup \{t\}$;
$\langle \Pi^r_{\mathcal{P}}, I^r_{\mathcal{P}} \rangle, P^r, Q^r :=$ ConstructRegionGraph($\langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle, P, Q$);
$\mathit{IsQRemoved} := \mathit{false}$;
$\langle V, A \rangle :=$ ConstructMaxDelayGraph($R(\mathcal{P})$);
$\langle V', A' \rangle :=$ ConstructSubgraph($\langle V, A \rangle, P^r, Q^r, \tau$); {Phase 2}
$T^r_{\mathcal{P}'} := \{(r_1, r_2) \in T^r_{\mathcal{P}} \mid (\gamma(r_1), \gamma(r_2)) \in A' \vee$
$\exists r_0 : \mathit{Weight}(\gamma(r_0), \gamma(r_1)) = 1 - \epsilon\}$;
$(\exists r_0 \in \mathcal{S}^r_{\mathcal{P}} \;:\; (\forall r_1 \in \mathcal{S}^r_{\mathcal{P}} \;:\; (r_0, r_1) \notin T^r_{\mathcal{P}'}))$ {Phase 3} $(r_0 \in Q^r)$
$\mathit{IsQRemoved} := \mathit{true}$;
$Q^r := Q^r - \{r_0\}$;
$T^r_{\mathcal{P}} := T^r_{\mathcal{P}} - \{(r, r_0) \mid (r, r_0) \in T^r_{\mathcal{P}}\}$;
break;
$(r_0 \notin I^r_{\mathcal{P}})$
$T^r_{\mathcal{P}'} := T^r_{\mathcal{P}'} - \{(r, r_0) \mid (r, r_0) \in T^r_{\mathcal{P}'}\}$;
`declare that addition is not possible`;
exit();
$(\mathit{IsQRemoved} = \mathit{false})$;
ConstructRealTimeProgram($\langle \Pi^r_{\mathcal{P}'}, I^r_{\mathcal{P}'} \rangle$); {Phase 4}

- Arcs of weight $c' - c - \epsilon$ from v_0 to v_1, where $c, c' \in \mathbb{Z}_{\geq 0}$, $c' > c$, and $0 < \epsilon \ll 1$, if (1) $\gamma^{-1}(v_0)$ is a boundary region with respect to clock variable x, (2) $\gamma^{-1}(v_1)$ is an open region whose time-successor $\gamma^{-1}(v_2)$ is a boundary region with respect to clock variable x, (3) $\gamma^{-1}(v_0) \to \gamma^{-1}(v_1)$ represents a delay transition in $R(\mathcal{P})$, and (4) $\gamma^{-1}(v_0).x = c$ and $\gamma^{-1}(v_2).x = c'$.

- Self-loop arcs of weight ∞ at vertex v, if $\gamma^{-1}(v)$ is an end region.

In order to compute the maximum time delay between region predicates P^r and Q^r, it suffices to find the longest distance between $\Gamma(P^r)$ and $\Gamma(Q^r)$ in G. In our addition algorithm, the procedure ConstructMaxDelayGraph transform a region graph $R(\mathcal{P}) = \langle \Pi^r_{\mathcal{P}}, I^r_{\mathcal{P}} \rangle$ into a MaxDelay digraph $G = \langle V, A \rangle$ as a black box.

The addition algorithm. We now describe the algorithm Add_rtUNITY in details (see Algorithm 6.1):

- (*Phase 1*) First, in order to keep track of time elapsed since P has become true

Procedure 6.2 ConstructSubgraph

Input: $\mathsf{MaxDelay}$ digraph $\langle V, A \rangle$, set of vertices V_p and V_q , and an **integer** τ.
Output: a subgraph $\langle V', A' \rangle$.

1: $V' := V$;
2: $A' := \{\}$; **each** vertex v in V_p the length of the shortest path Π from v to V_q is at most τ
3: $A' := A' \cup \{a \mid a \text{ is on } \Pi\}$;
4: $A' := A' \cup \{(u, v) \in A \mid (\forall w \in V' : (w, u) \notin A') \ \vee \ (u \in V_q)\}$;
5: $\langle V', A' \rangle$;

in a computation, we add an extra clock variable t to $X_{\mathcal{P}}$ and reset it on immediate transitions whose source state is not in P and target state is in P (Lines 1-2). Next, we construct the region graph $R(\mathcal{P}) = \langle \Pi^r_{\mathcal{P}}, I^r_{\mathcal{P}} \rangle$ (Line 3). We now reduce our problem to the problem of bounding the length of the longest path in ordinary weighted digraphs. Towards this end, we first generate the $\mathsf{MaxDelay}$ digraph $\langle V, A \rangle$ (Line 6).

- (*Phase 2*) Next, we invoke the procedure $\mathsf{ConstructSubgraph}$ (Line 7) which takes a $\mathsf{MaxDelay}$ digraph $\langle V, A \rangle$, an integer τ, and two sets of vertices V_p and V_q as input and generates a subgraph of $\langle V, A \rangle$, namely $\langle V', A' \rangle$, whose length of longest path from every vertex in V_p to V_q is bounded by τ (see Procedure 6.2). We begin with an empty set of arcs (Line 2). Next, we include arcs that participate in the shortest path from each vertex in V_p to a vertex in V_q provided the length of the path is at most τ (Lines 3-7). Then, we add the rest of the arcs to $\langle V', A' \rangle$ (Line 8) except the ones that originate from a shortest path from V_p to V_q identified in Lines 3-7. After invoking $\mathsf{ConstructSubgraph}$, we transform $\langle V', A' \rangle$ back into a region graph $R(\mathcal{P}') = \langle \Pi^r_{\mathcal{P}'}, I^r_{\mathcal{P}'} \rangle$ (Line 8 in Algorithm 6.1).

- (*Phase 3*) We now remove deadlock regions (created due to pruning of arcs in Phase 2) from $R(\mathcal{P}')$ (Lines 9-22). However, we need to consider a special case where a region r_0 in Q^r becomes a deadlock region (Lines 10-15). In this case, it is possible that all the regions along a path that starts from some region, say r, in P^r and end in r_0 become deadlock regions. Hence, our algorithm needs to identify a new path from r to a region in Q^r other than r_0. Thus, in such a case, we remove r_0 from Q^r (Lines 12-13) and rerun the algorithm from scratch. If an initial region becomes a deadlock region, we declare failure (Lines 18-20).

- (*Phase 4*) Finally, we construct and return a real-time program out of the region graph $R(\mathcal{P}')$ with revised set of edges $T^{r}_{\mathcal{P}'}$ (Lines 24).

Level of non-determinism. In order to increase the level of non-determinism, we may include additional paths whose length is at most τ. However, every time we add a path, we need to test whether or not this path creates new paths of length greater than τ. To this end, we can use one of the algorithms in the literature of graph theory (e.g., [Epp99]) to find and add the k shortest paths in an ordinary weighted digraph.

Theorem 6.1.1 *The algorithm* Add_rtUNITY *is sound and complete.*

Proof. In order to prove soundness, we show that the outcome of the algorithm meets the constraints of Problem Statement 3.2.1. Since we do not add new states or transitions, constraints $C1$-$C3$ are trivially satisfied. Moreover, by construction, the algorithm only includes states in P from where all computations can reach a state in Q within τ time units. Finally, since the algorithm removes deadlock regions, it does not introduce new time-convergent behaviors to the input program. Therefore, the output of the algorithm satisfies constraint $C4$ as well.

In order to prove the completeness, we show that if an initial state becomes a deadlock state, this state is deadlocked in all real-time programs that satisfy the constraints of Problem Statement 3.2.1. Observe that the only states that our algorithm may make unreachable are states in P from where there does not exist a computation that reaches a state in Q within τ. Clearly, such states cannot be present in any program that satisfies the constraints of Problem Statement 3.2.1. Moreover, if a state, say s_1, in P becomes unreachable by removing all its incoming transitions, it is possible that some other state, say s_0, becomes a deadlock state. Likewise, such a state cannot be present in any program that satisfies the constraints of Problem Statement 3.2.1. If s_0 is an initial state then our algorithm declares failure. Notice that in this case, there exists no solution to the Problem Statement 3.2.1. ∎

Theorem 6.1.2 *The complexity of* Add_rtUNITY *algorithm is polynomial-time in the size of the input program's region graph.* ∎

Proof. Observe that the algorithm Add_rtUNITY first generates the region graph of the input program. Then, it performs a sequence of polynomial-time procedures (e.g., reachability analysis and finding shortest paths) in the size of the region graph. Hence, the complexity of our algorithm is in polynomial-time in the size of the input program's region graph. ∎

6.1.1 Example: Real-Time Resource Allocation

We now demonstrate how the algorithm Add_rtUNITY works using an example on a real-time resource allocation program. The program $\mathcal{RA}$ consists of two processes j, where $j \in \{1, 2\}$. Each process needs two steps to complete and each step needs 1 time unit to complete. In the first step, the process submits a request for a resource. In the second step, the process performs an I/O operation using the acquired resource. Also, only one step is being executed at a time. The *timed actions* of $\mathcal{RA}$ are as follows:

$$
\begin{array}{llcl}
\mathcal{RA}1_j :: & req.j \;\wedge\; (x = 1) & \xrightarrow{\{x\}} & io.j,\; req.j \;:=\; true,\; false; \\
\mathcal{RA}2_j :: & io.j \;\wedge\; (x = 1) & \xrightarrow{\{x\}} & req.j,\; io.j \;:=\; true,\; false; \\
\mathcal{RA}3 :: & 0 \leq x \leq 1 & \longrightarrow & \mathbf{wait};
\end{array}
$$

where $j \in \{1, 2\}$. Clearly, in $\mathcal{RA}$, each process may keep acquiring a resource and performing I/O operation by an unbounded time duration. However, we would like to ensure that process $j = 1$ performs its I/O operation within 2 time units after acquiring the resource. To this end, we add the bounded-time leads-to property $\mathcal{L} \equiv (io.1 \mapsto_2 req.1)$. Based on what the algorithm Add_rtUNITY prescribes, we first need to add a new clock variable t and reset it whenever $io.1$ becomes true. Moreover, we let $c_t = 2$ when generating the region graph (see Figure 6.1). Next, we add the shortest path (the bold edges in Figure 6.1) from each region where $io.1$ becomes true to a region where $req.1$ holds. Subsequently, we can add additional k shortest paths (the zigzag edges in Figure 6.1) that preserve $\mathcal{L}$. It is easy to see that the algorithm Add_rtUNITY prunes dotted edges in Figure 6.1. Also, the regions shown in dotted circles are made unreachable by Add_rtUNITY due to removal of the dotted edges. Thus, the timed guarded commands of the revised program are as follows:

$$
\begin{array}{llcl}
\mathcal{RA}1_1 :: & req.1 \;\wedge\; (x = 1) & \xrightarrow{\{x,t\}} & io.1, req.1 \;:=\; true, false; \\
\mathcal{RA}2_1 :: & io.1 \;\wedge\; (x = 1) & \xrightarrow{\{x\}} & req.1, io.1 \;:=\; true, false; \\
\mathcal{RA}1_2 :: & req.2 \;\wedge\; (x = 1) \;\wedge\; (io.1 \Rightarrow t \leq 1) & \xrightarrow{\{x\}} & io.2, req.2 \;:=\; true, false; \\
\mathcal{RA}2_2 :: & io.2 \;\wedge\; (x = 1) \;\wedge\; (io.1 \Rightarrow t \leq 1) & \xrightarrow{\{x\}} & req.2, io.2 \;:=\; true, false;
\end{array}
$$

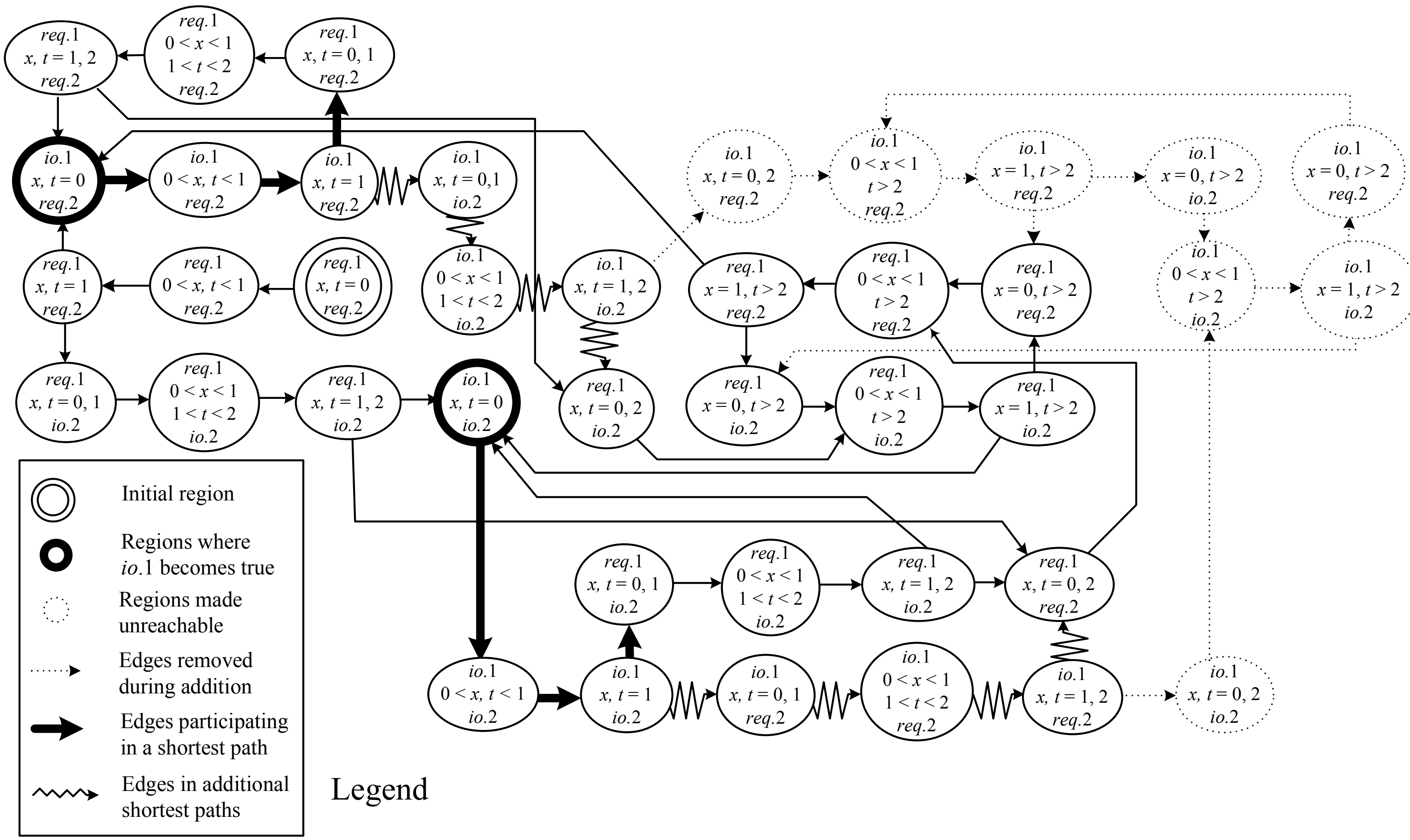


Figure 6.1: Region graph of the real-time resource allocation program.

$\mathcal{RA}3$: $\quad 0 \leq x \leq 1 \quad \longrightarrow \quad$ **wait**;

Notice that if we only add the shortest paths from regions where $io.1$ becomes true, i.e., we do not add additional shortest paths, then in the resulting program, the second conjunct in timed actions $\mathcal{RA}1_2$ and $\mathcal{RA}2_2$ would be replaced with $io.1 = \mathit{false}$. In this case, we would force the program to always execute the second step of process $j = 1$ (i.e., timed action $\mathcal{RA}2_1$) immediately after the first step (i.e., timed action $\mathcal{RA}1_1$).

We refer the reader to [BK06a] for another example on revising a controller for a railroad crossing gate which originally exhibits unbounded wait.

6.2 Revising Real-Time UNITY Programs with Maximum Non-determinism

In this section, we show that the problem of adding a bounded-time leads-to property to a real-time program while maintaining maximum non-determinism is NP-complete in the size of the program's region graph even if the given program satisfies the corresponding unbounded leads-to property.

Instance. Region graph $R(\mathcal{P}) = \langle \Pi^r_{\mathcal{P}}, I^r_{\mathcal{P}} \rangle$ of a real-time program $\mathcal{P}$, a bounded-time leads-to property $\mathcal{L} \equiv (P \mapsto_\tau Q)$, and a positive integer k, where $\mathcal{P}$ satisfies $P \mapsto Q$ and $|T^r_{|} \geq k$.

The decision problem (MNBL). Given the above instance, does there exist a region graph $R(\mathcal{P}') = \langle \Pi^r_{\mathcal{P}'}, I^r_{\mathcal{P}'} \rangle$, such that $|T^r_{\mathcal{P}'}| \geq k$ and $R(\mathcal{P}')$ meets the constraints of Problem Statement 3.2.1?

We now prove that MNBL is NP-complete by a reduction from the *vertex splitting problem* [PRS94, PRS98] in weighted directed acyclic graphs (DAG). This problem is described next.

The DAG Vertex Splitting Problem (DVSP). Let $G = \langle V, A \rangle$ be a weighted DAG and v_{sc}, v_{tg} be unique source and target vertices in V where the indegree of v_{sc} and the outdegree of v_{tg} are zero. Let G/Y denote the DAG when each vertex $v \in Y$ is split into vertices v^{in} and v^{out} such that all arcs $(v, u) \in A$, where $u \in V$, are replaced by arcs of the form (v^{out}, u) and all arcs $(w, v) \in A$, where $w \in V$, are replaced by arcs of the form (w, v^{in}). In other words, the outgoing arcs of v now leave vertex v^{out} while the incoming arcs of v now enter v^{in}, and, there is no arc between v^{in} and v^{out}. The DAG vertex splitting problem is to find a vertex set Y, where $Y \subseteq V$ and a positive integer i, where $|Y| \leq i$, such that the length of the longest path of G/Y from v_{sc} to v_{tg} is bounded by a pre-specified value d. DVSP is known to be NP-complete [PRS94, PRS98], for the case where $d \geq 2$ and the weight of all arcs is 1.

Theorem 6.2.1 *The problem of adding a bounded-time* leads-to *property to a real-time program is NP-complete in the size of the program's region graph even if the program satisfies the corresponding unbounded* leads-to *property.*

Proof. Since membership to NP is trivial, we show that the problem is NP-hard.

Mapping. Let $G = \langle V, A \rangle$ be any instance of DVSP whose longest path is to be bounded by d. We construct now a real-time program $\mathcal{MP}$ (and as a result the region graph $R(\mathcal{MP}) = \langle \Pi^r_{\mathcal{MP}}, I^r_{\mathcal{MP}} \rangle$) in the form of timed guarded commands as follows. Let the set of discrete and clock variables of $\mathcal{MP}$ be the singletons $\{l\}$ and $\{x\}$, respectively. For each vertex $v \in V - \{v_{tg}\}$ and for the target vertex v_{tg}, we introduce the following timed guarded commands:

$$
\begin{array}{llllll}
\mathcal{MP}_{v.1} :: & (l = v^{in}) & \wedge\ (x = 0) & \wedge\ (l \neq v^{in}_{tg}) & \longrightarrow & l := v^{out}; \\
\mathcal{MP}_{v.2} :: & (l = v^{in}) & \wedge\ (x = 0) & \wedge\ (l \neq v^{in}_{tg}) & \longrightarrow & l := v^{out}_{tg}; \\
\mathcal{MP}_{v_{tg.n.1}} :: & (l = v^{in}_{tg}) & \wedge\ (x = 0) & & \longrightarrow & l := tmp_{tg.n}; \\
\mathcal{MP}_{v_{tg.n.2}} :: & (l = tmp_{tg.n}) & \wedge\ (x = |A| + 1 - d) & & \xrightarrow{\{x\}} & l := v^{out}_{tg}; \\
\mathcal{MP}_{v_{tg}} :: & (l = v^{out}_{tg}) \vee & & & & \\
 & ((l = tmp_{tg.n}) & \wedge\ (0 \leq x \leq |A| + 1 - d)) & & \longrightarrow & \mathbf{wait};
\end{array}
$$

for all $1 \leq n \leq 2|V|$. For the source vertex v_{sc}, we let $v^{in}_{sc} = v^{out}_{tg}$. Also, for each arc (u, v) in A, we introduce the following timed guarded commands to $\mathcal{MP}$:

$$
\begin{array}{lllll}
\mathcal{MP}_{(u,v).j.1} :: & (l = u^{out}) & \wedge\ (x = 0) & \longrightarrow & l := tmp_{(u,v).j}; \\
\mathcal{MP}_{(u,v).j.2} :: & (l = tmp_{(u,v).j}) & \wedge\ (x = 1) & \xrightarrow{\{x\}} & l := v^{in}; \\
\mathcal{MP}_{(u,v)} :: & (l := tmp_{(u,v).j}) & \wedge\ (0 \leq x \leq 1) & \longrightarrow & \mathbf{wait};
\end{array}
$$

for all j, where $1 \leq j \leq 2|V|$. Intuitively, for each arc $(u, v) \in A$, the discrete variable l in program $\mathcal{MP}$ is assigned one of the following values: v^{in}, v^{out}, u^{in}, u^{out}, or $tmp_{(u,v).1} \cdots tmp_{(u,v).2|V|}$. Clearly, the value of l along with the clock regions identify $\mathcal{S}^r_{\mathcal{MP}}$. The set of initial regions of $R(\mathcal{MP})$ is the singleton $I^r_{\mathcal{MP}} = \{(l = v^{in}_{sc}, x = 0)\}$. The set $T^r_{\mathcal{MP}}$ of edges is identified by the above set of timed guarded commands. Finally, we let $P = \{s \mid l(s) = v^{in}_{sc}\}, Q = \{s \mid l(s) = v^{out}_{tg}\}$, $k = |T^r_{\mathcal{MP}}| - i$, and $\tau = |A| + 1$. Observe that all computations of $\mathcal{MP}$ start from where P holds and eventually reach Q, as G is acyclic. Hence, $\mathcal{MP}$ satisfies $P \mapsto Q$. We note that since v_{sc} and v_{tg} are unique vertices in G, Q is reachable from all states in $\mathcal{MP}$ and, hence, $\mathcal{MP}$ satisfies $true \mapsto Q$ as well.

Reduction. We need to show that a vertex $v \in Y$ in G must be split iff the corresponding timed guarded command $(l = v^{in}) \wedge (x = 0) \wedge (l \neq v^{in}_{tg}) \rightarrow l := v^{out}$ must be removed from $\mathcal{MP}$:

- $(\Rightarrow)$ Let the answer to DVSP be the set Y, where $|Y| \leq i$, i.e., after splitting all vertices $v \in Y$, the length of the longest path in G is at most d. We obtain the region graph $R(\mathcal{MP}') = \langle \Pi^r_{\mathcal{MP}'}, I^r_{\mathcal{MP}'} \rangle$ as follows. First, we let $\mathcal{S}^r_{\mathcal{MP}'} = \mathcal{S}^r_{\mathcal{MP}}$ and $I^r_{\mathcal{MP}'} = I^r_{\mathcal{MP}}$. In order to obtain $T^r_{\mathcal{MP}'}$, we remove the edges that correspond to timed action $\mathcal{MP}_{v.1}$ from $T^r_{\mathcal{MP}}$, for all $v \in Y$. Since v_{tg} is the unique target vertex in G, Q remains to be the set $\{s \mid l(s) = v^{out}_{tg}\}$ in $\mathcal{MP}'$. Thus, any computation of $\mathcal{MP}'$ that begins from a state in P will reach Q. Now, we show that the maximum time delay to reach Q is τ. Observe that there are two ways to reach Q: (1) from the state where $l = v^{in}_{tg}$ (using timed actions $\mathcal{MP}_{v_{tg}.n.1}$ and $\mathcal{MP}_{v_{tg}.n.2}$ for some n, $1 \leq n \leq 2|V|$), and (2) from a state where $l \neq v^{in}_{tg}$ (using the immediate transition in timed action $\mathcal{MP}_{v.2}$ for some $v \in V$). In the former case, the delay in reaching the state where $l = v^{in}_{tg}$ is less than d and since the time delay of timed actions $\mathcal{MP}_{v_{tg}.n.1}$ and $\mathcal{MP}_{v_{tg}.n.2}$ is $|A|+1-d$, the total time delay to reach Q is at most $|A|+1 = \tau$. In the latter case, by construction, the delay to reach Q is at most τ. Moreover, recall that $k = |T^r_{\mathcal{MP}}| - i$. Therefore, $\mathcal{MP}'$ meets the constraints of Problem Statement 3.2.1 with respect to $\mathcal{L}$ and $|T^r_{\mathcal{MP}'}| \geq |T^r_{\mathcal{MP}}| - i = k$.

- $(\Leftarrow)$ Let the answer to MNBL be $R(\mathcal{MP}') = \langle \Pi^r_{\mathcal{MP}'}, I^r_{\mathcal{MP}'} \rangle$, where $|T^r_{\mathcal{MP}'}| \geq k$ and the maximum delay to reach Q from P is at most τ. Note that $I^r_{\mathcal{MP}'}$ must be $\{(l = v^{in}_s, x = 0)\}$. Observe that in order to obtain $R(\mathcal{MP}')$, removing one or more timed guarded commands $\mathcal{MP}_{(u,v).j.1}$, $\mathcal{MP}_{(u,v).j.2}$, $\mathcal{MP}_{v_{tg}.n.1}$, or $\mathcal{MP}_{v_{tg}.n.2}$ does not contribute in bounding the maximum delay. This is due to the fact that the number of edges removed from $T^r_{\mathcal{MP}}$ is at most $|T^r_{\mathcal{MP}}| - k$, and $k = |T^r_{\mathcal{MP}}| - i$, where $i \leq |V|$, and there are $2|V|$ of such guarded commands and, hence, their removal would not change the maximum delay. Thus, we can assume that the edges removed are of the form $(l = v^{in}) \wedge (x = 0) \wedge (l \neq v^{in}_{tg}) \rightarrow l := v^{out}$. Observe that in order to reach Q from P, a computation either takes a timed guarded command $\mathcal{MP}_{v.2}$ for some $v \in V$, or it reaches Q via the state where $l = v^{in}_{tg}$. Clearly, in the later case, the delay to reach the state where $l = v^{in}_{tg}$ is at most $\tau - (|A|+1-d) = d$. In the former case, the corresponding path in G does not exist and, hence, its length does not matter. Thus, the timed actions removed to obtain $T^r_{\mathcal{MP}'}$ identify the set Y of vertices that should be split in G, i.e, $Y = \{v \in V - \{v_{tg}\} \mid ((l = v^{in}, x = 0) \rightarrow (l = v^{out}, x = 0)) \in (T^r_{\mathcal{MP}} - T^r_{\mathcal{MP}'})\}$. ∎

6.3 Adding Interval-Bounded Leads-to Properties

In this section, we consider automatic addition of *interval bounded-time* leads-to properties to real-time programs. An interval bounded-time leads-to property is of the form $\mathcal{L}_I \equiv (P \mapsto_{\leq[\delta_1,\delta_2]} Q)$, where $\delta_1 > 0$. This property expresses computations where if P becomes true then Q must becomes true within δ_2 time units but not earlier than δ_1 time units. As a naive solution, let us apply the algorithm Add_rtUNITY to add $\mathcal{L}_I$ to a real-time program. Since the required response time has a lower bound, the subroutine ConstructSubgraph has to first enumerate and ignore all the paths that start from P and reach Q such that their length is less than δ_1. Obviously, this enumeration cannot be accomplished in polynomial-time in the size of the region graph. In fact, we show that the problem of adding an interval bounded-time leads-to property to a real-time program is NP-complete by a simple reduction from the longest path problem [GJ79].

Instance. Region graph $R(\mathcal{P}) = \langle \Pi^r_{\mathcal{P}}, I^r_{\mathcal{P}} \rangle$ of a real-time program $\mathcal{P}$ and an interval bounded-time leads-to property $SPEC_n \equiv (P \mapsto_{\leq[\delta_1,\delta_2]} Q)$.

The decision problem (IBRA). Given the above instance, does there exist a region graph $R(\mathcal{P}') = \langle \Pi^r_{\mathcal{P}'}, I^r_{\mathcal{P}'} \rangle$ such that $R(\mathcal{P}')$ meets the constraints of the Problem Statement

3.2.1?

The longest path problem. Given are a directed weighted graph $G = \langle V, A \rangle$, a source vertex v_s, a target vertex v_t, and an integer k. The problem is to determine whether or not there exists a simple path in G, i.e., a sequence of distinct vertices $v_1, v_2, \ldots, v_m$, such that $v_1 = v_s$, $v_m = v_t$, and $\sum_{i=1}^{m-1} \mathit{Weight}(v_i, v_{i+1}) \geq k$, where $(v_i, v_{i+1}) \in A$ for all i, $1 \leq i \leq m-1$.

Theorem 6.3.1 *The problem of adding an interval bounded-time* leads-to *property to a real-time program is NP-complete in the size of the the program's region graph.*

Proof. First, we map an instance of the longest path problem to an instance of the IBRA problem as follows. Let the set of discrete and clock variables of $\mathcal{P}$ be the singletons $\{l\}$ and $\{x\}$, respectively. For each edge $(u, v) \in A$, we include the following timed guarded commands in $\mathcal{P}$:

$$\begin{array}{lllll} \mathcal{P}1_{(u,v)} & :: & (l = u) \ \wedge \ (x = \mathit{Weight}(u,v)) & \xrightarrow{\{x\}} & l := v; \\ \mathcal{P}2_{(u,v)} & :: & (l = u) \ \wedge \ (x \leq \mathit{Weight}(u,v)) & \longrightarrow & \mathbf{wait}; \end{array}$$

We also include the following delay action to ensure that all communications are infinite:

$$\mathcal{P}3 \ :: \ (l = v_t) \quad \longrightarrow \quad \mathbf{wait};$$

Moreover, we let $P = \{s \mid l(s) = v_s\}, Q = \{s \mid l(s) = v_t\}$, $\delta_2 = \infty$, and $\delta_1 = k$, where k is the bound on the length of the longest path in the instance of the longest path problem. Clearly, the value of l along with the clock regions identify $\mathcal{S}^r_{\mathcal{P}}\mathcal{P}$. We also let $I_{\mathcal{P}} = \{v_s\}$. Now, we show that G has a path of length of at least k from v_s to v_t iff there exists $R(\mathcal{P}') = \langle \Pi^r_{\mathcal{P}'}, I^r_{\mathcal{P}'} \rangle$ such that it meets the constraints of the Problem Statement 3.2.1. It is easy to see that if there exists a path in G from v_s to v_t whose length is greater than or equal to k then we can construct $R(\mathcal{P}')$ using the corresponding computation in $\mathcal{P}$ plus the delay action $\mathcal{P}3$. Moreover, if the answer to IBRA is affirmative then there exists a computation in $R(\mathcal{P}')$ which starts from P and reaches Q after δ_1 time units. Hence, the answer to the longest path problem is affirmative as well. ∎

Part III

Revising Programs in Open Systems

In Part II of this dissertation, we assumed that the program to be revised does not interact with the environment in any ways. In Parts III and IV of the dissertation, we consider *open systems* where programs do interact with the environment. To this end, we model such interactions via augmenting program computations with transitions that are uncontrollable by the program and cause unanticipated time delays, state corruptions, or clock resets. We refer to these uncontrollable transitions as *faults*. Thus, the notion of program revision in the context of open systems informally translates to synthesizing *fault-tolerant* programs as follows:

> Given are a *fault-intolerant* program $\mathcal{P}$, a set F of faults, and a specification $SPEC$, where $\mathcal{P}$ refines $SPEC$ in the absence of F, but $\mathcal{P}$ does not refine $SPEC$ in the presence of F. The problem is whether or not it is possible to automatically revise $\mathcal{P}$ inside the state space of $\mathcal{P}$ such that the revised program is *fault-tolerant* in the sense that $\mathcal{P}$ refines $SPEC$ in the absence and presence of F.

Throughout the dissertation, we refer to this problem as *synthesizing fault-tolerant programs* or *addition of fault-tolerance* and we consider various *levels of fault-tolerance* in order to study the problem.

This part is organized as follows. First, in Chapter 7, we present the type of safety and liveness properties that we consider in this part. We present our fault model and the notion of levels of fault-tolerance as well. We also formally state the problem of revising programs in open systems in Chapter 7. Following the problem statement, we present our results on the complexity of revising programs in open systems in Chapters 8-12. More specifically, in Chapter 8, we present our results on the complexity of transforming fault-intolerant real-time programs to fault-tolerant real-time programs. Chapter 9 is dedicated to the concept of synthesizing bounded-time phased recovery in fault-tolerant real-time programs. Chapter 10 studies a decomposition method for facilitating verification of (automatically synthesized or manually designed) fault-tolerant real-time programs. In Chapter 11, we present symbolic (BDD-based) heuristics for adding fault-tolerance to distributed programs. Finally, in Chapter 12, we present our tool SYCRAFT, which can transform a fault-intolerant distributed program to a fault-tolerant program.

Chapter 7

The Revision Problem in Open Systems

In this chapter, we formalize the problem of revising programs in open systems. This chapter is organized as follows. Section 7.1 is dedicated to present the type of specifications that we consider in revising programs in open systems. Then, in Section 7.2, we introduce our fault model and the notion of levels of fault-tolerance. Finally, in Section 7.3, we formally state the revision problem.

7.1 Basic Concepts

Since programs in open systems are subject to faults, they may reach states that are unreachable by program transitions alone. Thus, the notion of closure needs to be defined. Intuitively, by *closure* of a state predicate S in a transition predicate T, we mean that (1) if an immediate transition in T originates in S, then it must terminate in S as well, and (2) if a delay transition of T originates in S, then it must remain in S continuously.

Definition 7.1.1 **(closure)** A state predicate S is *closed* in transition predicate T iff

$$(\forall(\sigma_0, \sigma_1) \in T^s \ : \ ((\sigma_0 \in S) \ \Rightarrow \ (\sigma_1 \in S))) \ \wedge$$
$$(\forall(\sigma, \delta) \in T^d \ : \ ((\sigma \in S) \ \Rightarrow \ \forall \epsilon \in \mathbb{R}_{\geq 0} \mid (\epsilon \leq \delta) \ : \ \sigma + \epsilon \in S)). \blacksquare$$

Following Definitions 7.1.1 and 2.1.6, by closure of a program $\mathcal{P}$ in a state predicate S, we mean the closure of its transition predicate $T_{\mathcal{P}}$ in S. The concept of closure applies to processes in the obvious way.

In Chapter 2, we defined what we mean by a program refining a specification. In the context of open systems, it is crucial to augment the notion of closure in refinement. To this end, we introduce the concept of *satisfaction* which is stronger than refinement. Moreover, in order to reason about the correctness of programs in the absence of faults, we incorporate invariance properties of programs. Intuitively, an *invariant predicate* is one from where the execution of a program is closed and the program refines its specification. More specifically, we use invariant predicates to specify (1) closure properties, and (2) initial states of programs. Thus, given a program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$, in Parts III and IV, the state predicate $I_{\mathcal{P}}$ denotes an invariant predicate and we rename this predicate by $Inv_{\mathcal{P}}$. Below, we formally define the notions of specification *satisfaction* and *invariant*.

Definition 7.1.2 (satisfies) Let $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$ be a program and $SPEC$ be a specification for $\mathcal{P}$. We write $\mathcal{P} \models_{Inv_{\mathcal{P}}} SPEC$ and say that $\mathcal{P}$ *satisfies SPEC from* $Inv_{\mathcal{P}}$ iff (1) $Inv_{\mathcal{P}}$ is closed in $T_{\mathcal{P}}$, and (2) $\mathcal{P}$ refines $SPEC$. ∎

Definition 7.1.3 (invariant) Let $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$ be a program and $SPEC$ be a specification for $\mathcal{P}$. If $\mathcal{P} \models_{Inv_{\mathcal{P}}} SPEC$, we say that $Inv_{\mathcal{P}}$ is an *invariant predicate of* $\mathcal{P}$ *for* $SPEC$. ∎

Whenever the specification is clear from the context, we will omit it; thus, "$Inv_{\mathcal{P}}$ is an invariant of $\mathcal{P}$" abbreviates "$Inv_{\mathcal{P}}$ is an invariant of $\mathcal{P}$ for $SPEC$". Note that Definition 7.1.2 introduces the notion of satisfaction with respect to infinite computations. In case of finite computations, we characterize them by determining whether they can be extended to an infinite computation in the specification using the following definition.

Definition 7.1.4 (maintains) Let $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$ be a program and $SPEC$ be a specification for $\mathcal{P}$. We say that $\mathcal{P}$ *maintains SPEC* from $Inv_{\mathcal{P}}$ iff (1) $Inv_{\mathcal{P}}$ is closed in $T_{\mathcal{P}}$, and (2) for all computation prefixes $\overline{\alpha}$ of $\mathcal{P}$ that start from a state in $Inv_{\mathcal{P}}$, there exists a computation suffix $\overline{\beta}$ such that $\overline{\alpha}\overline{\beta} \in SPEC$. We say that $\mathcal{P}$ *violates SPEC* iff it is not the case that $\mathcal{P}$ maintains $SPEC$. ∎

We note that if $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$ satisfies $SPEC$ from $Inv_{\mathcal{P}}$, then $\mathcal{P}$ maintains $SPEC$ from $Inv_{\mathcal{P}}$ as well, but the reverse direction does not always hold. We introduced the notion of *maintains* for computations that a (fault-intolerant) program cannot produce, but the computation can be extended to one that is in *SPEC* by adding *recovery*. We now present the type of specifications we consider in Part III of the dissertation.

7.1.1 The Type of Specifications in Part III

In order to express time-related behaviors of real-time programs (e.g., deadlines and recovery time), we focus on a standard property typically used in real-time computing known as the *bounded-time response property* (or simply *bounded response*).

Definition 7.1.5 (bounded response property) Let P and Q be two state predicates and δ be a nonnegative integer. A bounded response property, denoted $P \mapsto_{\leq\delta} Q$, is the set of all computations $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ in which, for all $i \in \mathbb{Z}_{\geq 0}$, if $\sigma_i \in P$, then there exists j, $j \geq i$, such that (1) $\sigma_j \in Q$, and (2) $\tau_j - \tau_i \leq \delta$. In other words, it is always the case that a state in P is followed by a state in Q within δ time units. We call P the *event predicate*, Q the *response* (or *recovery*) *predicate*, and δ the *response* (or *recovery*) *time*. ∎

Similar to Part II, the specifications considered in Part III are an intersection of a *safety* specification and a *liveness* specification [AS85, Hen92]. In particular, we concentrate on a special case where the specification is the intersection of (1) *timing independent safety* characterized by a set of bad instantaneous transitions (denoted $SPEC_{\overline{bt}}$), (2) *timing dependent safety* characterized by a set of bounded response properties (denoted $SPEC_{\overline{br}}$), and (3) liveness (see Definition 2.2.5). Recall that we assume that the set of variables and their respective domains of a program and its specification are identical.

Definition 7.1.6 (safety specifications)

1. (*timing independent safety*) Let $SPEC_{bt}$ be a finite set of instantaneous *bad transitions* of the form $(s_0, \nu) \rightarrow (s_1, \nu[\lambda := 0])$, where s_0 and s_1 are two locations and λ is set of clock variables. We denote the specification whose computations have no transition in $SPEC_{bt}$ by $SPEC_{\overline{bt}}$.

2. (*timing constraints*) We denote $SPEC_{\overline{br}}$ by the conjunction $\bigwedge_{i=0}^{m}(P_i \mapsto_{\leq\delta_i} Q_i)$, where for all i, $0 \leq i \leq m$, P_i and Q_i are two state predicates, and $\delta_i \in \mathbb{Z}_{\geq 0}$. ∎

Thus, given a specification *SPEC*, one can implicitly identify $SPEC_{\overline{bt}}$ and $SPEC_{\overline{br}}$ as defined above. Throughout the paper, $SPEC_{\overline{br}}$ is meant to prescribe how a program should meet its timing constraints such as providing bounded-time recovery to its normal behavior after the occurrence of faults. We formally define the notion of recovery in Section 7.2.

As far as liveness specification, we require that our revision methods preserve liveness properties of the input program. Thus, liveness properties need not be specified explicitly.

7.1.2 Example

Real-time traffic controller. We presented the timing independent safety specification of the real-time traffic controller program in Subsection 2.2.1. One invariant predicate for $\mathcal{TC}$ is the following:

$$
\begin{aligned}
Inv_{\mathcal{TC}} = \forall i \in \{0,1\} \ :& \\
&[(sig_i = G) \ \Rightarrow \ ((sig_j = R) \ \wedge \ (x_i \leq 10) \ \wedge \ (z_i > 1))] \ \wedge \\
&[(sig_i = Y) \ \Rightarrow \ ((sig_j = R) \ \wedge \ (y_i \leq 2) \ \wedge \ (z_i > 1))] \ \wedge \\
&[((sig_i = R) \ \wedge \ (sig_j = R)) \\
&\qquad\qquad\quad \Rightarrow \ ((z_i \leq 1) \ \oplus \ (z_j \leq 1))],
\end{aligned}
$$

where $j = (i+1) \mod 2$ and $\oplus$ denotes the *exclusive-or* operator. It is straightforward to see that $\mathcal{TC}$ satisfies $SPEC_{\overline{bt}_{\mathcal{TC}}}$ (defined in Subsection 2.2.1) from $I_{\mathcal{TC}}$. We present the timing constraints of $\mathcal{TC}$ in Section 7.2.

Distributed Byzantine agreement. We presented the safety specification of the Byzantine agreement program $\mathcal{BA}$ in Subsection 2.2.1. One possible invariant predicate of $\mathcal{BA}$ consists of the following sets of states:

1. First, we consider the set of states where the general is non-Byzantine. In this case:

   - one of the non-general processes may be Byzantine,
   - if a non-general process, say j, is non-Byzantine, it is necessary that $d.j$ be initialized to either $\perp$ or $d.g$, and
   - a non-Byzantine process cannot finalize its decision if its decision equals $\perp$.

2. We also consider the set of states where the general is Byzantine. In this case, g can change its decision arbitrarily. It follows that other processes are non-Byzantine and $d.j, d.k$ and $d.l$ are initialized to the same value that is different from $\perp$.

Thus, the invariant predicate is as follows:

$$
\begin{aligned}
Inv_{\mathcal{BA}} = \neg b.g \ \wedge \ (\neg b.j \vee \neg b.k) \ \wedge \ (\neg b.k \vee \neg b.l) \ \wedge \ (\neg b.l \vee \neg b.j) \ \ \wedge \\
(\forall p \in \{j, k, l\} \ : \ \neg b.p \ \Rightarrow \ (d.p = \bot \ \vee \ d.p = d.g)) \ \wedge \\
(\forall p \in \{j, k, l\} \ : \ (\neg b.p \wedge f.p) \ \Rightarrow \ (d.p \neq \bot)) \ \vee \\
(b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l \wedge (d.j = d.k = d.l \ \wedge \ d.j \neq \bot))
\end{aligned}
$$

An alert reader can easily verify that $\mathcal{BA}$ satisfies $SPEC_{\overline{bt_{\mathcal{BA}}}}$ from $Inv_{\mathcal{BA}}$.

Notice that in case of both the traffic controller and Byzantine agreement programs, the invariant predicate is a superset of the initial states introduced in Subsections 2.1.2 and 2.1.3. This should clarify the double role of invariant predicates for specifying both closure properties and (initial) states starting from where the program satisfies its specification.

7.2 Fault Model and Fault-Tolerance

In this section, we generalize formal definitions of faults and fault-tolerance due to Arora and Gouda [AG93], and Kulkarni [Kul99].

7.2.1 Fault Model

The faults that a program is subject to are systematically represented by a transition predicate. Precisely, a class of *faults* F for program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$ is a transition predicate in the state space of program $\mathcal{P}$, i.e., $F \subseteq \mathcal{S}_{\mathcal{P}} \times \mathcal{S}_{\mathcal{P}}$. Thus, similar to program transitions, faults are classified by *immediate faults* and *delay faults*. We use $\mathcal{P}[]F$ to denote the program $\mathcal{P}$ in the presence of faults F. Hence, transitions of program $\mathcal{P}$ in the presence of F is obtained by taking the union of the transitions in $T_{\mathcal{P}}$ and the transitions in F.

We emphasize that such representation is possible notwithstanding the type of the faults (be they stuck-at, crash, fail-stop, timing, performance, Byzantine, message loss, etc.), the nature of the faults (be they permanent, transient, or intermittent), or the ability of the program to observe the effects of the faults (be they detectable or undetectable).

Just as we introduced the notion of invariant for reasoning about the correctness of programs in the absence of faults, we introduce the notion of *fault-span* to reason about program correctness in the presence of faults.

Definition 7.2.1 (fault-span) Let $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$ be a program and F be a set of faults. We say that a state predicate $S_{\mathcal{P}}$ is an F-span (read as *fault-span*) of $\mathcal{P}$ from $Inv_{\mathcal{P}}$

iff the following conditions are satisfied:

1. $Inv_{\mathcal{P}} \subseteq S_{\mathcal{P}}$, and
2. $S_{\mathcal{P}}$ is closed in $T_{\mathcal{P}} \cup F$. ∎

Observe that for all computations of $\mathcal{P}$ that start from states where $Inv_{\mathcal{P}}$ is true, $S_{\mathcal{P}}$ is a boundary in the state space of $\mathcal{P}$ up to which (but not beyond which) the state of $\mathcal{P}$ may be perturbed or delayed by the occurrence of the transitions in F. Subsequently, as we defined the computations of $\mathcal{P}$, one can define computations of program $\mathcal{P}$ in the presence of faults F by simply substituting $T_{\mathcal{P}}$ with $T_{\mathcal{P}} \cup F$ in Definition 2.1.7.

7.2.2 Levels of Fault-Tolerance

Obviously, in the absence of faults, a program should satisfy its specification. In the presence of faults, however, it may not satisfy the entire specification and, hence, it may satisfy some (possibly) weaker *tolerance specification*. These specifications are based on satisfaction of a combination of timing independent safety, liveness, timing constraints, and a desirable bounded-time recovery mechanism. Intuitively, we consider three levels of fault-tolerance, namely, *nonmasking*, *failsafe*, and *masking* based on satisfaction of timing independent safety (i.e., $SPEC_{\overline{bt}}$ in Definition 7.1.6) and liveness (i.e., recovery to the program invariant) properties in the presence of faults. For failsafe and masking fault-tolerance, we propose two additional levels, namely, *soft* and *hard*, based on satisfaction of timing constraints (i.e., $SPEC_{\overline{br}}$ in Definition 7.1.6) in the presence of faults.

In order to motivate the idea of soft and hard fault-tolerance, let us consider the railroad crossing problem. Suppose that a train is approaching a railroad crossing. The timing-independent safety specification requires that :

> "if the train is crossing, the gate must be closed."

Obviously, this requirement is not time-related. In addition to this requirement, a timing constraint requires that:

> "once the gate is closed, it should reopen within 5 minutes."

In this example, it may be catastrophic if the gate is open while the train is crossing due to occurrence of faults. To the contrary, if the gate remains closed for more than 5 minutes due

	Timing-independent safety ($SPEC_{\overline{bt}}$)	Bounded recovery to invariant	Timing constraints ($SPEC_{\overline{br}}$)
Soft-Failsafe	✓		
Hard-Failsafe	✓		✓
Nonmasking		✓	
Soft-Masking	✓	✓	
Hard-Masking	✓	✓	✓

Table 7.1: Levels of fault-tolerance.

to occurrence of faults, the outcome is undesirable, but not disastrous. Thus, depending upon the outcome of violation of a safety specification, the desired level of fault-tolerance varies. In the railroad crossing problem, we require that the system must tolerate faults that cause the gate to remain open while the train is crossing. We call such a system *soft fault-tolerant*. Intuitively, a soft fault-tolerant program is not required to satisfy its timing constraints in the presence of faults, but it has to meet its timing-independent safety properties in both absence and presence of faults.

Now, consider a system that controls internal pressure of a boiler. Suppose that in this system, the safety specification requires that:

> "once a pressure gauge reads 30 pounds per square inch, the controller must issue a command to open a valve within 20 seconds."

In such a system, if occurrence of faults causes the controller not to respond within the required time, the outcome may be disastrous. Thus, our boiler controller must satisfy its timing constraints even in the presence of faults. In other words, the boiler controller must be *hard fault-tolerant*. Intuitively, a hard fault-tolerant program must satisfy its timing constraints even in the presence of faults. In fact, in hard fault-tolerant programs, the demand for hard real-time processing merges with catastrophic consequences of systems, whereas in soft fault-tolerance the catastrophic consequences are not related to the program's timing constraints. Table 7.1 illustrates an informal description of the levels of fault-tolerance in the context of real-time programs based on satisfaction of $SPEC_{\overline{bt}}$, $SPEC_{\overline{br}}$, and bounded-time recovery.

We now formally define the levels of fault-tolerance. The strongest level of fault-tolerance is called *hard-masking*. A hard-masking program satisfies its entire specification in both absence and presence faults, i.e., it masks the occurrence of faults.

Definition 7.2.2 (hard-masking fault-tolerance) Let $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$ be a program, F be a set of faults, and $SPEC$ be a specification. We say that $\mathcal{P}$ is *hard-masking F-tolerant to $SPEC$ from $Inv_{\mathcal{P}}$ with recovery time δ, $\delta \in \mathbb{Z}_{\geq 0}$,* iff

1. $\mathcal{P} \models_{Inv_{\mathcal{P}}} SPEC_{\overline{bt}}$,
2. $\mathcal{P} \models_{Inv_{\mathcal{P}}} SPEC_{\overline{br}}$,
3. there exists a state predicate S such that S is an F-span of $\mathcal{P}$ from $Inv_{\mathcal{P}}$ and
   (a) $\mathcal{P}[]F$ maintains $SPEC_{\overline{bt}}$ from S,
   (b) $\mathcal{P}[]F$ maintains $SPEC_{\overline{br}}$ from S, and
   (c) every computation of $\mathcal{P}[]F$ that starts from a state in S, reaches a state in $Inv_{\mathcal{P}}$ within δ time units. ∎

We now define other levels of fault-tolerance by weakening Definition 7.2.2 based on satisfaction of $SPEC_{\overline{bt}}$, $SPEC_{\overline{br}}$, or bounded-time recovery in the presence of faults.

Definition 7.2.3 (weaker levels of fault-tolerance)

- *(Nonmasking)* We say that $\mathcal{P}$ is *nonmasking F-tolerant to $SPEC$ from $Inv_{\mathcal{P}}$ with recovery time δ, $\delta \in \mathbb{Z}_{\geq 0}$,* iff it meets conditions 1, 2, and 3-c of Definition 7.2.2.

- *(Soft-Failsafe)* We say that $\mathcal{P}$ is *soft-failsafe F-tolerant to $SPEC$ from $Inv_{\mathcal{P}}$* iff it meets conditions 1, 2, and 3-a of Definition 7.2.2.

- *(Hard-Failsafe)* We say that $\mathcal{P}$ is *hard-failsafe F-tolerant to $SPEC$ from $Inv_{\mathcal{P}}$* iff it meets conditions 1, 2, 3-a, and 3-b of Definition 7.2.2.

- *(Soft-Masking)* We say that $\mathcal{P}$ is *soft-masking F-tolerant to $SPEC$ from $Inv_{\mathcal{P}}$ with recovery time δ, $\delta \in \mathbb{Z}_{\geq 0}$,* iff it meets conditions 1, 2, 3-a, and 3-c of Definition 7.2.2. ∎

Notation. Given a program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$ and a set F of faults, whenever the specification $SPEC$ and the invariant $Inv_{\mathcal{P}}$ are clear from the context, we omit them; thus, "F-tolerant" abbreviates "F-tolerant to $SPEC$ from $Inv_{\mathcal{P}}$".

7.2.3 Example

Real-time traffic controller. Our real-time traffic controller program $\mathcal{TC}$ is subject to clock reset faults due to circuit malfunctions. In particular, we consider faults that reset either z_0 or z_1 at any state in the invariant predicate $Inv_{\mathcal{TC}}$ (cf. Subsection 7.1.2), without changing the location of $\mathcal{TC}$:

$$\begin{array}{lll} F_0 :: \ Inv_{\mathcal{TC}} & \xrightarrow{\{z_0\}} & \textbf{skip}; \\ F_1 :: \ Inv_{\mathcal{TC}} & \xrightarrow{\{z_1\}} & \textbf{skip}; \end{array}$$

It is straightforward to see that in the presence of F_0 and F_1, $\mathcal{TC}$ may violate $SPEC_{\overline{bt}_{\mathcal{TC}}}$. For instance, consider the case where F_1 occurs when $\mathcal{TC}$ is in a state of $Inv_{\mathcal{TC}}$ where $(sig_0 = sig_1 = R) \wedge (z_0 \leq 1) \wedge (z_1 > 1)$. In the resulting state, we have $(sig_0 = sig_1 = R) \wedge (z_0 \leq 1) \wedge (z_1 = 0)$. From this state, immediate execution of timed actions $\mathcal{TC}3_0$ and then $\mathcal{TC}3_1$ results in a state where $(sig_0 = sig_1 = G)$, which is clearly a violation of the safety specification. In Chapter 9, we present a revision algorithm for synthesizing a fault-tolerant version of $\mathcal{TC}$ which ensures satisfaction of safety and provides bounded-time *phased recovery* to $Inv_{\mathcal{TC}}$ in the presence the above faults. The phased recovery condition can be modeled by timing constraints in the safety specification as follows. We require $\mathcal{TC}$ to, first, ensure that nothing catastrophic happens and then recover to its normal behavior. Thus, the fault-tolerant version of $\mathcal{TC}$ has to, first, reach a state where both signals are red and subsequently recover to $Inv_{\mathcal{TC}}$ where exactly one signal turns green. Thus,

$$SPEC_{\overline{br}_{\mathcal{TC}}} \equiv (\neg Inv_{\mathcal{TC}} \mapsto_{\leq 3} Q_{\mathcal{TC}}) \ \wedge \ (\neg Inv_{\mathcal{TC}} \mapsto_{\leq 7} Inv_{\mathcal{TC}}),$$

where $Q_{\mathcal{TC}} = \forall i \in \{0,1\} \ : \ ((sig_i = R) \ \wedge \ (z_i > 1))$. The response times in $SPEC_{\overline{br}_{\mathcal{TC}}}$ are simply two arbitrary numbers for illustration.

Distributed Byzantine agreement. In the context of the Byzantine agreement problem, the fault transitions that affect a process, say j, of $\mathcal{BA}$ are as follows: (We include similar actions for k, l, and g)

$$\begin{array}{llll} F_0 & :: \ \neg b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l & \longrightarrow & b.j := true; \\ F_1 & :: \ b.j & \longrightarrow & d.j, f.j := 0|1, false|true; \end{array}$$

where $d.j := 0|1$ means that $d.j$ could be assigned either 0 or 1. In case of the general process, the second action does not change the value of any f-variable. It is easy to observe that $\mathcal{BA}$ may violate both *validity* and *agreement* (see Subsection 2.2.1) in the presence of F_0 and F_1. The program may also reach deadlock states. In Chapter 11, we present an algorithm for synthesizing fault-tolerant distributed programs and demonstrate its application in order to synthesize a fault-tolerant version of $\mathcal{BA}$ which ensures satisfaction of safety and liveness in the presence the above faults.

7.3 Problem Statement

Given are a fault-intolerant program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$, a set F of faults, and a specification $SPEC$ such that $\mathcal{P} \models_{Inv_{\mathcal{P}}} SPEC$. Our goal is to revise $\mathcal{P}$ in order to synthesize a program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$ such that $\mathcal{P}'$ is F-tolerant to $SPEC$ from $Inv_{\mathcal{P}'}$. We require that our revision methods obtain $\mathcal{P}'$ from $\mathcal{P}$ by *adding fault-tolerance* to $\mathcal{P}$ without introducing new behaviors in the absence of faults. To this end, we first define the notion of *projection*. Informally, projection of a transitions predicate T on a state predicate S consists of immediate transitions of T^s that start in S and end in S, and delay transitions of T^d that start and remain in S continuously.

Definition 7.3.1 (projection) *Projection* of a set T of transitions on a state predicate S (denoted $T|S$) is the following set of transitions:

$$
\begin{aligned}
T|S = \{(\sigma_0, \sigma_1) \in T^s \mid \sigma_0, \sigma_1 \in S\} \ \cup \\
\{(\sigma, \delta) \in T^d \mid \sigma \in S; \wedge\ (\forall \epsilon \in \mathbb{R}_{\geq 0} \mid (\epsilon \leq \delta)\ :\ \sigma + \epsilon \in S)\}. \ \blacksquare
\end{aligned}
$$

Likewise, the projection of a set of edges $T^r_{\mathcal{P}}$ on region predicate S^r (denoted $T^r_{\mathcal{P}}|U^r$) is the set of edges $\{(r_0, r_1) \in T^r_{\mathcal{P}} \mid r_0, r_1 \in U^r\}$.

Similar to the revision problem for closed systems, since meeting timing constraints in the presence of faults requires time predictability, we let our synthesis methods incorporate a finite set X_n of new clock variables. We denote the program obtained by removing the clock variables in Y from $X_{\mathcal{P}}$ by $\mathcal{P} \backslash Y$. Obviously, no state and transition predicate of $\mathcal{P} \backslash Y$ depend on the value of variables in Y.

Observe that in the absence of faults:

1. If $Inv_{\mathcal{P}'}$ contains states that are not in $Inv_{\mathcal{P}}$, then $\mathcal{P}'$ may include computations that start outside $Inv_{\mathcal{P}'}$. Since we require that $\mathcal{P}' \models_{Inv_{\mathcal{P}'}} SPEC$, it implies that $\mathcal{P}'$ is using a new way to satisfy $SPEC$. Therefore, we require that $(Inv_{\mathcal{P}' \setminus X_n}) \subseteq Inv_{\mathcal{P}}$.

2. Regarding the transitions of $\mathcal{P}$ and $\mathcal{P}'$, we focus on transitions that occur inside the invariant $Inv_{\mathcal{P}'}$. If $T_{\mathcal{P}'}|Inv_{\mathcal{P}'}$ contains a transition that is not in $T_{\mathcal{P}}|Inv_{\mathcal{P}'}$, then $\mathcal{P}'$ can use this transition in order to satisfy $SPEC$. Therefore, we require that $(T_{\mathcal{P}' \setminus X_n}|Inv_{\mathcal{P}' \setminus X_n}) \subseteq (T_{\mathcal{P}}|Inv_{\mathcal{P}' \setminus X_n})$.

Thus, the revision problem in the context of open systems is as follows.

Problem Statement 7.3.2 Given a program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$, specification $SPEC$, and a set F of faults such that $\mathcal{P} \models_{Inv_{\mathcal{P}}} SPEC$, identify $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$ such that:

($C1$) $\mathcal{S}_{\mathcal{P}' \setminus X_n} = \mathcal{S}_{\mathcal{P}}$, where X_n is a finite set of new clock variables,

($C2$) $(Inv_{\mathcal{P}' \setminus X_n}) \subseteq Inv_{\mathcal{P}}$,

($C3$) $(T_{\mathcal{P}' \setminus X_n}|Inv_{\mathcal{P}' \setminus X_n}) \subseteq (T_{\mathcal{P}}|Inv_{\mathcal{P}' \setminus X_n})$, and

($C4$) $\mathcal{P}'$ is F-tolerant to $SPEC$ from $Inv_{\mathcal{P}'}$. ■

Obviously, the problem statement can be instantiated for different levels of fault-tolerance introduced in Definitions 7.2.2 and 7.2.3. Notice that constraint $C3$ implicitly implies that the synthesized program is not allowed to exhibit new finite computations, which in turn means that synthesis algorithms are not allowed to create deadlock states. Put it another way, this condition ensures that if the given intolerant program satisfies a *universally quantified property*, then the synthesized fault-tolerant program satisfies it as well.

Comparison to controller synthesis and game theory. Our formulation of the synthesis problem is in spirit close to both controller synthesis, where program and fault transitions may be modeled as controllable and uncontrollable actions, and game theory, where program and fault transitions may be modeled in terms of two players. In fact, in both problems, the objective is to *restrict* the program actions at each state through synthesizing a controller or a wining strategy such that the behavior of the entire system is always *desirable* according to safety and reachability conditions, in the presence of an adversary. Notice that the conditions $C1$ and $C2$ precisely express this notion of restriction.

Furthermore, the conjunction of all conditions expresses the notion of *language inclusion*, where the synthesized program is supposed to exhibit a subset of behaviors of the input intolerant program. Unlike controller synthesis and game theory algorithms where arbitrary specifications are often considered, our algorithms are tailored for properties typically used in specifying real-time and fault-tolerance requirements and, hence, they synthesize programs more efficiently. We elaborate on comparison and contrast with controller synthesis and game theory in Chapter 15 in detail.

Chapter 8

Synthesizing Real-Time Fault-Tolerant Programs

In this chapter, we study the problem of addition of fault-tolerance to real-time fault-intolerant programs. Recall that in Chapter 7, we defined the notion of fault-tolerance in terms of a variety of levels based on satisfaction of timing independent safety, timing constraints, and liveness specifications in the presence of faults. For each level of fault-tolerance, in this chapter, we present the complexity of its addition to an input fault-intolerant program.

The rest of this chapter is organized as follows. First, in Section 8.1, we present the Altitude Switch program, which we use as a running demonstration throughout this chapter. In Section 8.2, we show that there exists a polynomial-time sound and complete algorithm for adding nonmasking fault-tolerance to a given fault-intolerant program in the size of the intolerant program's region graph. Then, in Section 8.3, we present a polynomial-time sound and complete algorithm for adding soft-failsafe fault-tolerance to a given fault-intolerant program in the size of the intolerant program's region graph. We also show the same result for adding hard-failsafe fault-tolerance, where the timing dependent safety specification consists of only one timing constraint. Finally, in Section 8.4, we present our results on automated synthesis of soft and hard-masking fault-tolerant programs. Specifically, we present a polynomial-time sound and complete algorithm for adding soft-masking fault-tolerance to a given program. Then, we show that the problem of adding hard-masking fault-tolerance to a given program where the safety specification requires two or more timing constraints is NP-complete.

8.1 Case Study: Altitude Switch

We use a simplified version of an altitude switch program [BH00] as a running example to illustrate the concepts and algorithms presented in this chapter.

Fault-intolerant program

The altitude switch program (denoted $\mathcal{AS}$) reads a set of input variables. Then, it powers on a Device of Interest (Doi) when the aircraft descends below a pre-specified altitude threshold. There exist three internal variables, a mode variable that determines the operating mode of the program, two environment variables, and three watchdog timers (for purpose of synthesis the variables are treated identically):

- The internal variables are as follows: (i) *iAltBelow* is equal to 1 if the altitude is below a pre-specified threshold, otherwise, it is equal to 0; (ii) *iDOIStatus* is equal to 1 if the Doi is powered on, otherwise, it is equal to 0, and (iii) *iCorruptSensor* is equal to 1 if the system reads an invalid value from the altitude meter sensors, otherwise, it is equal to 0.

- The $\mathcal{AS}$ program can be in three different modes: (i) initialization mode when the system is initializing; (ii) await mode when the system is waiting for the Doi to power on, and (iii) standby mode. We use the variable *mStatus* with domain $\{\textsc{Init}, \textsc{Await}, \textsc{Standby}\}$ to show the system modes in the program.

- We model the validity of signals that come from the altitude meter by the environment variable *eSensorReading*. This variable is equal to $\perp$ when the system fails to read the altitude meter. Otherwise, it is not equal to $\perp$ (i.e., it is equal to the altitude of the aircraft). The system may issue reset commands in different situations. The environment variable *eReset* is equal to 0 if a system reset command has been issued. Otherwise, it is equal to 1.

- $\mathcal{AS}$ has three watchdog timers: (i) the clock variable x measures the time elapsed since the system has been in the Init mode; (ii) the clock variable y measures the time elapsed since the system has failed to read the altitude meter (i.e., *iCorruptSensor* $=$

$$
\begin{array}{lll}
\mathcal{AS}1 :: (mStatus = \textsc{Init}) \ \wedge \ (x \leq 1) & \longrightarrow & mStatus := \textsc{Standby}; \\
\mathcal{AS}2 :: (mStatus = \textsc{Standby}) \ \wedge & & \\
\qquad (eReset = 0) & \xrightarrow{\{x\}} & mStatus, eReset := \textsc{Init}, 1; \\
\mathcal{AS}3 :: (mStatus = \textsc{Standby}) \ \wedge \ (iAltBelow = 0) \ \wedge & & \\
\qquad (iDOIStatus = 0) & \xrightarrow{\{z\}} & mStatus, iAltBelow := \textsc{Await}, 1; \\
\mathcal{AS}4 :: (mStatus = \textsc{Await}) \ \wedge & & \\
\qquad (iDOIStatus = 0) \ \wedge \ (z \leq 2) & \longrightarrow & mStatus, iDOIStatus := \textsc{Standby}, 1; \\
\mathcal{AS}5 :: (mStatus = \textsc{Await}) \wedge & & \\
\qquad (eReset = 0) \ \wedge \ (z \leq 2) & \xrightarrow{\{x\}} & mStatus, eReset := \textsc{Init}, 1; \\
\mathcal{AS}6 :: ((mStatus = \textsc{Standby}) \wedge (iCorruptSensor = 0)) \ \vee & & \\
\qquad (x \leq 1) \ \vee (y \leq 2) \ \vee \ (z \leq 2) & \longrightarrow & \textbf{wait}; \\
\mathcal{AS}7 :: (eSensorReading = \perp) \ \wedge & & \\
\qquad (iCorruptSensor = 0) & \xrightarrow{\{y\}} & iCorruptSensor := 1; \\
\mathcal{AS}8 :: (mStatus = \textsc{Standby}) \ \wedge & & \\
\qquad (iCorruptSensor = 1) \ \wedge \ (y \leq 2) & \xrightarrow{\{x\}} & mStatus, iCorruptSensor := \textsc{Init}, 0;
\end{array}
$$

Figure 8.1: Timed guarded commands of $\mathcal{AS}$.

1), and (iii) the clock variable z measures the time elapsed since the system has been in the AWAIT mode.

Formally, the timed guarded commands of $\mathcal{AS}$ are illustrated in Figure 8.1. The program changes its mode from INIT to STANDBY within 1 second (timed action $\mathcal{AS}1$). Also, the program may go back to the INIT mode if it is in STANDBY mode and the reset signal is received (timed action $\mathcal{AS}2$). If the program is in the STANDBY mode and the DOI is not powered on then the program *may* go to a state where the altitude of the aircraft is below the threshold and it is in the AWAIT mode (timed action $\mathcal{AS}3$). In this case, the program starts the watchdog timer z. Otherwise, the program stays in the STANDBY mode for an arbitrary time as long as the altitude meter is not showing an invalid value (delay action $\mathcal{AS}6$). In the AWAIT mode, the program either (i) powers on the DOI within 2 seconds and goes back to the STANDBY mode (timed action $\mathcal{AS}4$), or (ii) goes to the INIT mode upon receiving the reset signal (timed action $\mathcal{AS}5$). The program may take delays as long as as it does not violate the timing constraints of the program (delay action $\mathcal{AS}6$). If the program receives a signal indicating that the altitude meter sensors are showing an invalid value, it sets the variable *iCorruptSensor* to 1 and starts the timer y (timed action $\mathcal{AS}7$). In this case, the program should go back to the INIT mode within 2 seconds (timed action $\mathcal{AS}8$).

Invariant

The invariant of $\mathcal{AS}$ consists of the following states:

$$S_{\mathcal{AS}} = \{\sigma \mid \; ((mStatus(\sigma) = \textsc{Init}) \;\Rightarrow\; (x(\sigma) \leq 1)) \wedge$$
$$((mStatus(\sigma) = \textsc{Standby}) \;\Rightarrow\; (iCorruptSensor(\sigma) = 0 \vee y(\sigma) \leq 2)) \wedge$$
$$((mStatus(\sigma) = \textsc{Await}) \;\Rightarrow\; (z(\sigma) \leq 2))\}.$$

Safety specification

The safety specification requires that the program does not change its mode from Standby to Await, if it fails to read a valid altitude. Also, when the program is in a state outside the invariant then it can only go to the Init mode. Furthermore, the safety specification requires three timing constraints with respect to each watchdog timer. Formally, the safety specification is as follows:

$$SPEC_{bt} = \{(\sigma_0, \sigma_1) \mid$$
$$((iCorruptSensor(\sigma_0) = 1) \;\wedge\; (mStatus(\sigma_0) = \textsc{Standby}) \wedge$$
$$(mStatus(\sigma_1) = \textsc{Await})) \;\vee\; ((\sigma_0 \notin S_{\mathcal{AS}}) \;\wedge\; ((mStatus(\sigma_1) = \textsc{Standby}) \;\vee$$
$$(mStatus(\sigma_1) = \textsc{Await}))) \;\vee\; ((\sigma_0 \notin S_{\mathcal{AS}}) \;\wedge\; (eReset(\sigma_1) = 1))\}$$
$$SPEC_{br_1} = (mStatus = \textsc{Init}) \quad \mapsto_{\leq 1} \quad (mStatus = \textsc{Standby})$$
$$SPEC_{br_2} = (mStatus = \textsc{Await}) \quad \mapsto_{\leq 2} \quad (((mStatus = \textsc{Standby}) \wedge (iDOIStatus = 1)) \;\vee\; (mStatus = \textsc{Init}))$$
$$SPEC_{br_3} = ((mStatus = \textsc{Standby}) \;\wedge\; (iCorruptSensor = 1)) \mapsto_{\leq 2} (mStatus = \textsc{Init})$$

Region graph

Figure 8.2 illustrates the region graph of the $\mathcal{AS}$ program. Note that due to the large state space of $\mathcal{AS}$, we only include regions, edges, and (inside each region) variables that are

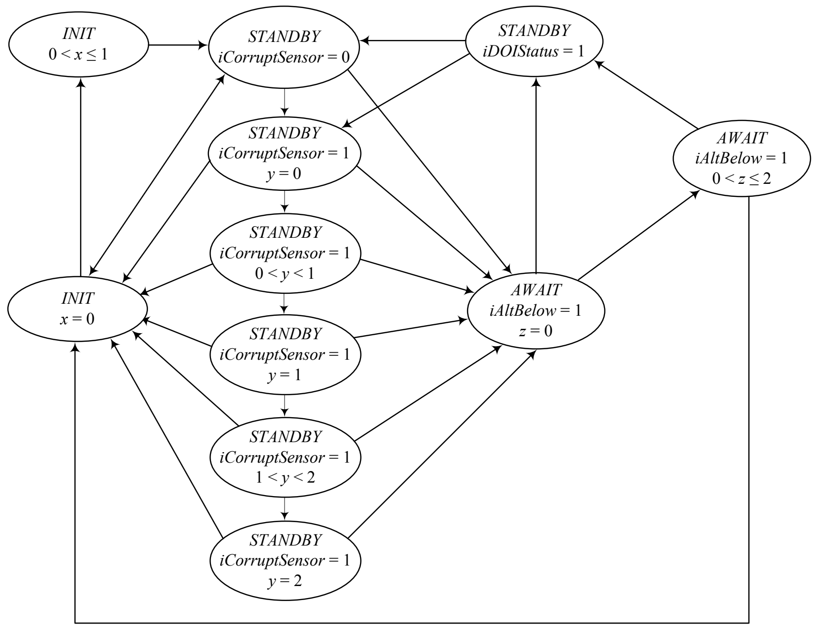


Figure 8.2: Region graph of $\mathcal{AS}$.

important to illustrate how our synthesis algorithms work in Sections 8.2, 8.3, and 8.4. For the same reason, some regions are collapsed into one region.

Faults

$\mathcal{AS}$ is subject to a set of delay faults where the program is (i) initializing; (ii) in Standby mode and the altitude meter shows an invalid value, or (iii) waiting for the Doi to power on. Moreover, immediate faults can corrupt the reading of the altitude meter (cf. Figure 8.3). Figure 8.4 illustrates the region graph of the $\mathcal{AS}$ program with respect to the mentioned delay faults. In sections 8.2, 8.3, and 8.4 we show that how we add nonmasking, hard-failsafe, and soft-masking fault-tolerance to $\mathcal{AS}$, respectively.

$$
\begin{array}{lcl}
F1 :: (\mathit{mStatus} = \textsc{Init}) \ \wedge \ (1 \leq x \leq 2) & \xrightarrow{\{t\}} & \textbf{wait}; \\
F2 :: (\mathit{mStatus} = \textsc{Standby}) \ \wedge & & \\
\qquad (\mathit{iCorruptSensor} = 1) \ \wedge \ (2 \leq y \leq 3) & \xrightarrow{\{t\}} & \textbf{wait}; \\
F3 :: (\mathit{mStatus} = \textsc{Await}) \ \wedge \ (2 \leq z \leq 3) & \xrightarrow{\{t\}} & \textbf{wait}; \\
F4 :: (\mathit{eSensorReading} \neq \bot) & \longrightarrow & \mathit{eSensorReading} := \bot;
\end{array}
$$

Figure 8.3: Fault timed actions in $\mathcal{AS}$.

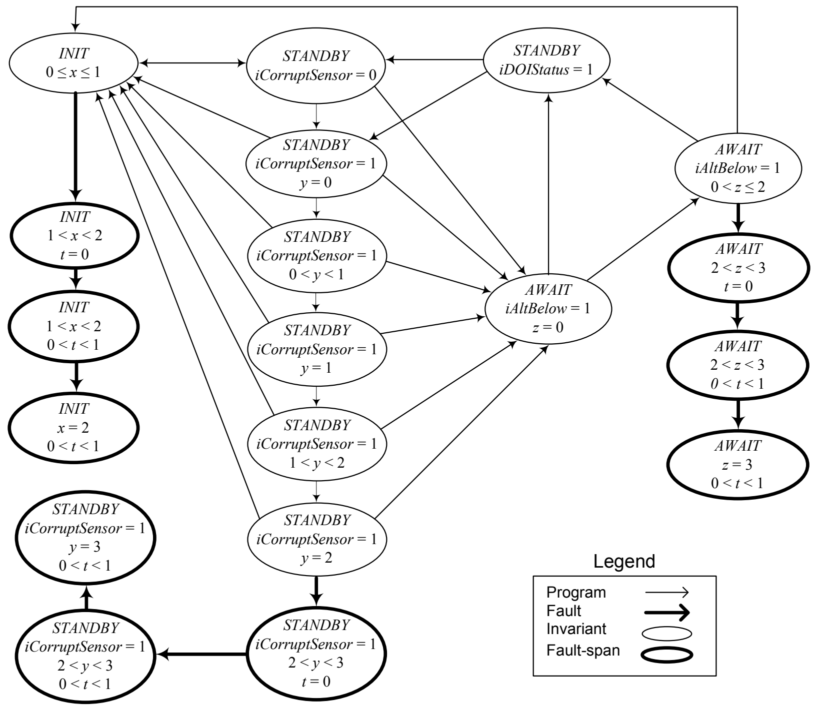


Figure 8.4: Region graph with respect to fault-span of $\mathcal{AS}$.

8.2 Adding Nonmasking Fault-Tolerance

In this section, we present a synthesis algorithm for adding nonmasking fault-tolerance to a given fault-intolerant program $\mathcal{P}$. Intuitively, to derive a nonmasking fault-tolerant program $\mathcal{P}'$, we need to ensure that if the state of $\mathcal{P}'$ is perturbed or delayed by faults then it recovers to a state in the invariant within a pre-specified recovery time. To this end, we first introduce the procedure Add_BoundedRecovery, which guarantees bounded-time recovery from a source state predicate, say P, to a target state predicate, say Q, within a required recovery time, say δ, in the presence of faults. Obviously, such recovery mechanism can be expressed as the bounded response property $P \mapsto_{\leq \delta} Q$. Thus, more formally, our goal is to ensure that the synthesized program $\mathcal{P}'$ satisfies $P \mapsto_{\leq \delta} Q$ even in the presence of faults. We reuse this algorithm in this section as well as in Sections 8.3 and 8.4 to design synthesis algorithms for adding nonmasking, hard-failsafe, and soft-masking fault-tolerance, respectively.

Before we describe our synthesis algorithms, we make the following assumptions in the context of real-time fault-tolerant programs

Assumption 8.2.1 We assume that faults are immediately detectable and that given a state of the program, we can determine the number of faults that have occurred in reaching that state. This assumption is needed only for addition of hard fault-tolerance and is realistic in many commonly-considered systems. For instance, in multiprocessor scheduling theory, a processor-crash is immediately detectable and its number of occurrences is easily traceable. Note that one can model faults whose detection is subject to time delays by a pair of fault transitions: first a delay fault and then the original fault transition. Based on this assumption, we augment fault timed actions with a timer t that is reset upon occurrence. ∎

Assumption 8.2.2 Observe that the above formulation of program computations in the presence of faults guarantees that the number of occurrence of faults in all computations is finite. However, since we deal with real-time programs and our goal is to design transformation algorithms that guarantee *bounded-time* recovery in the presence of faults, we assume that the number of occurrence of faults in all computations is bounded by some integer $n \in \mathbb{Z}_{\geq 0}$. This assumption is reasonable in many commonly-considered fault-tolerant real-time programs. In fact, it can be trivially shown that providing bounded-time recovery in the presence of unbounded number of arbitrary faults is not possible. ∎

Assumption 8.2.3 Let $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$ be a program that satisfies specification $SPEC$ from $Inv_{\mathcal{P}}$ in the absence of faults. Thus, since $\mathcal{P}$ satisfies $SPEC_{\overline{br}} \equiv \bigwedge_{i=0}^{m}(P_i \mapsto_{\leq \delta_i} Q_i)$, without loss of generality, we assume that for each bounded response property $(P_i \mapsto_{\leq \delta_i} Q_i)$, $1 \leq i \leq m$, $\mathcal{P}$ already has a clock variable that is reset on transitions, say (σ_0, σ_1) that originate where $\sigma \notin P_i$ and $\sigma_1 \in P_i$. This clock variable acts as a timer in order to keep track of time when P_i becomes true. ∎

8.2.1 Adding Bounded-Time Recovery in the Presence of Faults

Algorithm sketch. We now develop a procedure that adds bounded-time recovery to a given region graph from an arbitrary state predicate P to another state predicate Q within δ time units. The procedure has four phases. First, we transform the region graph into a weighted directed graph (called MaxDelay digraph [CY91]), in which the length of a path from a source vertex v_s to a target vertex v_t is equal to the the delay for reaching the region that corresponds to v_t from the region that corresponds to v_s. We use this digraph to identify and exclude the computations that violate $P \mapsto_{\leq \delta} Q$. Thus, in Phase 2, we rank vertices of the MaxDelay digraph by applying an *adjusted Dijkstra's shortest*

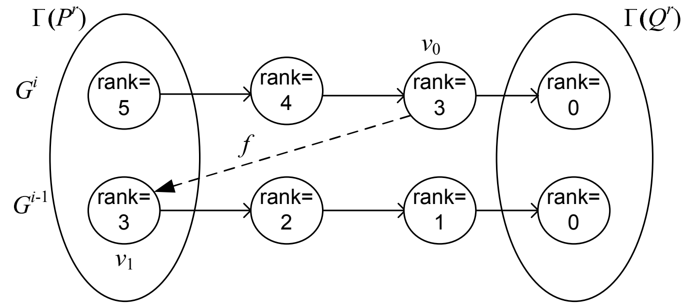


Figure 8.5: Adjusted shortest path.

path algorithm. This algorithm adjusts the length of the shortest path from a source to a target vertex by taking the amount of time *wasted* by occurrence of faults into account. For instance, suppose that a computation starts from a state $\sigma_0 \in P$. Now, if faults take the computation to a state where some computation prefix should be *redone*, the maximum delay of that computation to reach Q is obviously increased. Hence, we adjust the length of the shortest path from σ_0 to Q such that the amount of time wasted by every occurrence of faults is considered (cf. Figure 8.5 for an example). In Phase 3, we include regions and edges whose rank is at most the required response time δ. Finally, in Phase 4, we transform the synthesized MaxDelay digraph back into a region graph.

We note that the transformation of region graphs to MaxDelay digraphs is precisely the same as what described in Section 6.1. Our notation also follows the notation in Section 6.1. In particular, recall that γ is a bijection that maps each region in $R(\mathcal{P})$ to its corresponding vertex in the MaxDelay digraph. Also, Γ is a bijection that maps each region predicate $R(\mathcal{P})$ to its corresponding set of vertices in the MaxDelay digraph.

We now describe the procedure Add_BoundedRecovery (cf. Procedure 8.1) in detail:

- (*Phase 1*) Given a region graph $R(\mathcal{P})$, we first transforms it into a MaxDelay digraph $G = \langle V, A \rangle$ (Line 1). Recall that, by Assumption 8.2.1, faults are detectable and $\mathcal{P}$ has a variable that shows how many faults have occurred in a computation. Thus, we let $\langle V^i, A^i \rangle$ be the portion of G, in which $n - i$ faults have occurred, where $0 \leq i \leq n$ (Line 2). More specifically, initially, a computation starts from the portion G^n where no faults have occurred. If a fault occurs in a computation that is currently in portion G^i, the computation will proceed in portion G^{i-1}. We use these portions to verify whether it is possible to reach a vertex in $\Gamma(Q^r)$ from each vertex in $\Gamma(P^r)$ within δ time units.

Procedure 8.1 Add_BoundedRecovery

Input: region graph $R(\mathcal{P}) = \langle \Pi^r_{\mathcal{P}}, \mathit{Inv}^r_{\mathcal{P}} \rangle$, set of fault edges F^r, region predicates P^r, Q^r, and **integers** n and δ

Output: revised region graph $\langle \Pi^r_{\mathcal{P}'}, \mathit{Inv}^r_{\mathcal{P}'} \rangle$ by adding bounded-time recovery from P^r to Q^r in the presence of F^r, and, the set ns of regions from where $P \mapsto_{\leq \delta} Q$ may be violated in the presence of faults.

1: $\langle V, A \rangle :=$ ConstructMaxDelayGraph($R(\mathcal{P}) = \langle \Pi^r_{\mathcal{P}}, \mathit{Inv}^r_{\mathcal{P}} \rangle, F^r$); {Phase 1}
2: Let $G^i = \langle V^i, A^i \rangle$ be the portion of G, in which $(n - i)$ faults have occurred, where $0 \leq i \leq n$;
each vertex $v \in V^0$ {Phase 2}
3: Rank(v) := Length of the standard shortest path from v to $\Gamma(Q^r)^0$;
$i = 1$ **to** n
each vertex $v_0 \in V^i$:
4: $V_f := \{v_1 \in V^{i-1} \mid (\gamma^{-1}(v_0), \gamma^{-1}(v_1)) \in F^r\}$; $V_f \neq \{\}$
5: $\mathit{MinRank}(v_0) := \max\{(\mathrm{Rank}(v_1) + \mathit{Weight}(v_0, v_1))$ for all $v_1 \in V_f\}$; $\mathit{MinRank}(v_0) := 0$;
6: AdjustShortestPaths($\langle V^i, A^i \rangle, \Gamma(Q^r)^i$);
{*Constructing a subgraph of each portion such that the longest distance between* $\Gamma(P^r)$ *and* $\Gamma(Q^r)$ *is at most* δ *and then adding the arcs and vertices that do not appear on paths from* $\Gamma(P^r)$ *to* $\Gamma(Q^r)$}

$i = 0$ **to** n {Phase 3.1}
7: $\langle V'^i, A'^i \rangle := \{\}$; each vertex $v \in \Gamma(P^r)^i$ Rank(v) $\leq \delta$
8: $\Pi :=$ the shortest path from v to $\Gamma(Q^r)^i$;
9: $V'^i := V'^i \cup \{u \mid u \text{ is on } \Pi\}$;
10: $A'^i := A'^i \cup \{a \mid a \text{ is on } \Pi\}$;
11: $A'^i := A'^i \cup \{(u, v) \in A^i \mid u \notin V'^i \ \vee \ (u \in \Gamma(Q^r)^i)\}$;
12: $V'^i := V'^i \cup \{u \mid \exists v : ((u, v) \in A'^i \ \vee \ (v, u) \in A'^i)\}$;
13: Add additional paths from $\Gamma(P^r)$ to $\Gamma(Q^r)$; (see Remark 8.2.4); {Phase 3.2}
{*Transforming weighted digraph* G' *into a region graph*}
14: $T'^r_{\mathcal{P}} := \{(r_1, r_2) \in T^r_{\mathcal{P}} \mid ((\gamma(r_1), \gamma(r_2)) \in A') \ \vee \exists r_0 : \mathit{Weight}(\gamma(r_0), \gamma(r_1)) = 1 - \epsilon\}$; {Phase 4}
15: $ns := \{r \mid \gamma(r) \in (V - V')\}$;
16: **return** $T'^r_{\mathcal{P}}, ns$;

Procedure 8.2 AdjustShortestPaths

Input: directed weighted graph $G^i = \langle V^i, A^i \rangle$: and set of vertices V_q {*Adjusts the rank of each vertex based on the ranks computed in Add_BoundedRecovery*}

1: Apply Dijkstra's shortest path on all source vertices $v \in V^i$ to sink vertices V_q with the following modification: Dijkstra's shortest path computes a length less than $\mathit{MinRank}(v)$
2: Rank(v) := $\mathit{MinRank}(v)$;
3: Rank(v) := length of Dijkstra's shortest path from v to V_q;

- (*Phase 2*) Next, we rank vertices of all portions of G using a modified Dijkstra's shortest path algorithm, which takes state perturbations and delay faults into account (lines 3-15 in Procedure 8.1 and Procedure 8.2). More specifically, since no faults occur in G^0, we first let the rank of each vertex $v \in V^0$ be the length of standard Dijkstra's shortest path from v to $\Gamma(Q^r)^0$ (Line 4). Now, let v_0 be a vertex in V^i where $1 \leq i \leq n$, and let v_1 be a vertex in V^{i-1}, such that $(\gamma^{-1}(v_0), \gamma^{-1}(v_1))$ is a fault edge in $R(\mathcal{P})$ where v_0 is on a path from $\Gamma(P^r)$ to $\Gamma(Q^r)$. There exist two cases: (1) the fault edge $(\gamma^{-1}(v_0), \gamma^{-1}(v_1))$ decreases or does not change the computation delay, i.e, the length of the shortest distance from v_1 to $\Gamma(Q^r)^{i-1}$ is less than or equal to the length of the shortest distance from v_0 to $\Gamma(Q^r)^i$, and (2) the fault edge $(\gamma^{-1}(v_0), \gamma^{-1}(v_1))$ increases the computation delay, i.e., the length of the shortest distance from v_1 to $\Gamma(Q^r)^{i-1}$ is greater than the length of the shortest distance from v_0 to $\Gamma(Q^r)^i$ (cf. Figure 8.5 for an example). While the former case does not cause violation of $P \mapsto_{\leq \delta} Q$ in the presence of faults, the later may do. Hence, we set the rank of $v_0 \in V^i$ to at least the rank of $v_1 \in V^{i-1}$. Moreover, if there exist multiple fault edges at $\gamma^{-1}(v_0)$ (Line 8) then we take the maximum rank (Line 10). If there exist no fault edges at region $\gamma^{-1}(v_0)$, we temporarily let the rank of v_0 be 0 (Line 11). After computing the rank of vertices from where faults may occur, we adjust the rank of the rest of vertices from where faults do not occur by invoking the procedure AdjustShortestPath (Line 12).

- (*Phase 3*) Now, for each portion G^i, we construct a subgraph of G^i whose longest distance from each vertex in $\Gamma(P^r)^i$ to $\Gamma(Q^r)^i$ is at most δ (Lines 17-27). We begin with an empty digraph $\langle V'^i, A'^i \rangle$ and we first include the shortest paths from each vertex $v \in \Gamma(P^r)^i$ to $\Gamma(Q^r)^i$, provided $\text{Rank}(v) \leq \delta$ (Lines 19-23). Next, we include the remaining arcs and vertices in G'^i, so that no arcs of the form (v_0, v_1), where v_0 is on a path from $\Gamma(P^r)^i$ to $\Gamma(Q^r)^i$ are added (Lines 25-26). Note that if we add an arc that originates at a vertex on a path from $\Gamma(P^r)^i$ to $\Gamma(Q^r)^i$, we cannot guarantee that the resultant new path reaches $\Gamma(Q^r)^i$ within δ.

Remark 8.2.4 Observe that since we do not include all legitimate paths from $\Gamma(P^r)$ to $\Gamma(Q^r)$, the resultant program is not maximal in terms of transitions. However, one can include additional paths (which are not necessarily shortest paths) as long as their addition do not violate the given bounded response property (Phase 3.2). Such addition can be achieved using graph theoretic polynomial time algorithms for finding k-shortest paths (e.g., [Epp99]).

- (*Phase 4*) In the last phase, we first transform the digraph G' back into a region graph (Line 29). Then, we return the set $T'^r_{\mathcal{P}}$ of edges from where $P \mapsto_{\leq \delta} Q$ may not be violated even in the presence of faults, and the set *ns* of regions from where $P \mapsto_{\leq \delta} Q$ may be violated in the presence of faults (Lines 30-31).

8.2.2 Adding Nonmasking Fault-Tolerance Using Bounded-Time Recovery

In order to synthesize a nonmasking program, it only suffices to add bounded-time recovery from the fault-span to the invariant of the intolerant program, as a nonmasking program is not required to satisfy its safety specification in the presence of faults.

Algorithm sketch. Intuitively, our algorithm for adding nonmasking fault-tolerance consists of three main phases. In Phase 1, we transform the intolerant program into its region graph. Then, in Phase 2, we recompute the set of program transitions, invariant, and fault-span in a loop so that: (1) the fault-span is closed in the set of program transitions in the presence of faults, (2) program invariant is deadlock-free and it is a subset of the fault-span, and (3) from each state in the fault-span, there exists a path of length at most the desirable recovery time which ends at a state in the invariant. This loop terminates when a fixpoint in recomputing the fault-span and invariant is reached. Finally, in Phase 3, we transform the resultant region graph from Phase 2 back into a real-time program.

Algorithm 8.3 Add_Nonmasking

Input: real-time program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, \mathit{Inv}_{\mathcal{P}} \rangle$, transition predicate F, **integers** n, δ

Output: a nonmasking fault-tolerant program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, \mathit{Inv}_{\mathcal{P}'} \rangle$ with bounded-time recovery δ.

1: $\langle \Pi^r_{\mathcal{P}}, \mathit{Inv}^r_{\mathcal{P}} \rangle, F^r :=$ ConstructRegionGraph$(\langle \Pi_{\mathcal{P}}, \mathit{Inv}_{\mathcal{P}} \rangle, F)$; {Phase 1}
2: $S^r_1 := \mathcal{S}^r_{\mathcal{P}}$;
3: **repeat** {Phase 2}
4: $\quad S^r_2, \mathit{Inv}^r_2 \ := \ S^r_1, \mathit{Inv}^r_1$;
5: $\quad T^r_{\mathcal{P}_1} := T^r_{\mathcal{P}} | \mathit{Inv}^r_1 \ \cup \ \{((s_0, \rho_0), (s_1, \rho_1)) \mid (s_0, \rho_0) \in (S^r_1 - \mathit{Inv}^r_1) \ \wedge \ (s_1, \rho_1) \in S^r_1 \ \wedge \ \exists \rho_2 \mid \rho_2$ is a time-successor of $\rho_0 \ : \ (\exists \lambda \subseteq X \ : \ \rho_1 = \rho_2[\lambda - \{t\} := 0])\}$;
6: $\quad T^r_{\mathcal{P}_1}, ns :=$ Add_BoundedRecovery$(\langle \Pi^r_{\mathcal{P}_1}, \mathit{Inv}^r_{\mathcal{P}_1} \rangle, F^r, S^r_1 - \mathit{Inv}^r_1, \mathit{Inv}^r_1, n, \delta)$;
7: $\quad S^r_1 :=$ ConstructFaultSpan$(S^r_1 - ns, \ F^r)$;
8: $\quad \mathit{Inv}^r_1 :=$ RemoveDeadlocks$(\mathit{Inv}^r_1 \ \cap \ S^r_1, T^r_{\mathcal{P}_1})$;
9: $\quad$ **if** $(\mathit{Inv}^r_1 = \{\} \ \vee \ S^r_1 = \{\})$ **then**
10: $\quad\quad$ `declare no nonmasking` F`-tolerant program` $\mathcal{P}'$ `exists;`
11: $\quad\quad$ **exit**();
12: $\quad$ **end if**
13: **until** $(S^r_1 = S^r_2 \ \wedge \ \mathit{Inv}^r_1 = \mathit{Inv}^r_2)$;
14: $\langle \Pi_{\mathcal{P}'}, \mathit{Inv}_{\mathcal{P}'} \rangle :=$ ConstructRealTimeProgram$(\langle \Pi^r_{\mathcal{P}_1}, \mathit{Inv}^r_{\mathcal{P}_1} \rangle)$; {Phase 3}

We now describe the algorithm Add_Nonmasking (cf. Algorithm 8.3) in detail. Let $\mathcal{P} = \langle \Pi_{\mathcal{P}}, \mathit{Inv}_{\mathcal{P}} \rangle$ be a fault-intolerant program and F be a set of faults. In order to derive a nonmasking program $\mathcal{P}'$ with recovery time δ from $\mathcal{P}$, we merely need to add recovery from fault-span $S_{\mathcal{P}}$ to the invariant $\mathit{Inv}_{\mathcal{P}}$ within δ. In other words, adding nonmasking fault-tolerance reduces to adding the bounded response property $S_{\mathcal{P}} \mapsto_{\leq \delta} \mathit{Inv}_{\mathcal{P}}$ in the presence of F. Detailed steps of the algorithm are as follows:

- (*Phase 1*) We first transform the real-time program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, \mathit{Inv}_{\mathcal{P}} \rangle$, invariant S, and the set of fault transitions f into a region graph $R(\mathcal{P}) = \langle \Pi^r_{\mathcal{P}}, \mathit{Inv}^r_{\mathcal{P}} \rangle$, region invariant Inv^r_1, and fault edges F^r by invoking the procedure ConstructRegionGraph (Line 1), as described in Subsection 2.3. Next, we let our initial estimate of region fault-span be the entire region space (Line 2).

- (*Phase 2*) Next, we recompute the set $T^r_{\mathcal{P}_1}$ of edges, region fault-span S^r_1, and region invariant Inv^r_1 until we reach a fixpoint as follows (Lines 3-13):

  1. In order to compute the set $T^r_{\mathcal{P}_1}$ of edges (Line 5), we first include edges in $T^r_{\mathcal{P}}|Inv^r_1$. Then, in order to ensure that Inv^r_1 is reachable from all regions in S^r_1, we add *recovery edges* that originate from regions in $S^r_1 - Inv^r_1$ and terminate at regions in S^r_1 where the time-monotonicity condition is preserved. Notice that the algorithm allows arbitrary clock resets during recovery (except the clock variable t that keeps track of recovery time δ). Since such clock resets occur only in states outside the invariant, where a nonmasking program is not required to satisfy its safety specification, they are legitimate. After adding recovery edges, we invoke the procedure Add_BoundedRecovery (Line 6). This invocation identifies the set $T^r_{\mathcal{P}_1}$ of legitimate program edges with respect to our recovery time requirement, and the set *ns* of regions in S^r_1 from where recovery within δ is not possible.

  2. Since a program does not have control over occurrence of faults, the set of regions from where *ns* is reachable by taking fault edges alone should not be reachable. Thus, we recompute the region fault-span by invoking the procedure Construct-FaultSpan (see Procedure 8.4) with parameter $S^r_1 - ns$ (Line 7). This procedure itself is a largest fixpoint computation. In particular, in procedure Construct-FaultSpan, we keep removing the regions from where closure of fault-span is violated through a fault edge until no such regions exist.

  3. Next, due to the possibility of removal of some regions and edges in the previous steps, the algorithm removes deadlock regions by invoking the procedure RemoveDeadlocks (see Procedure 8.5), which is also a largest fixpoint calculation (Line 8). Notice that since Inv^r_1 must be a subset of S^r_1, we invoke the procedure

---

Procedure 8.4 ConstructFaultSpan

Input: region predicate S^r and set of edges F^r
Output: the largest subset of S^r that is closed in F^r

while $(\exists r_0, r_1 : r_0 \in S^r \wedge r_1 \notin S^r \wedge (r_0, r_1) \in F^r)$ **do**
  $S^r := S^r - \{r_0\};$
end while
return S^r;

Procedure 8.5 RemoveDeadlocks

Input: region predicate Inv^r and set of edges T^r

Output: the largest subset of Inv^r from where all computations of T^r are infinite

while $(\exists r_0 \in Inv^r \ : \ (\forall r_1 \mid (r_1 \neq r_0 \ \wedge \ r_1 \in Inv^r) \ : \ (r_0, r_1) \ \notin \ T^r))$ **do**
  $Inv^r := Inv^r - \{r_0\};$
end while
return Inv^r;

RemoveDeadlocks for $Inv_1^r \cap S_1^r$. In the procedure RemoveDeadlocks, we require that self-loops at open regions are not considered as outgoing edges. We will use this requirement to prove that computations of the synthesized program are time-divergent.

- (*Phase 3*) Finally, upon reaching a fixpoint, the algorithm transforms the synthesized region graph $R(\mathcal{P}_1) = \langle \Pi_{\mathcal{P}}^r, Inv_{\mathcal{P}}^r \rangle$ into a real-time program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$ (Line 14).

Theorem 8.2.5 *The algorithm* Add_Nonmasking *is sound and complete.*

Proof. First, we show that the algorithm is sound. To this end, we need to prove that the algorithm meets the three constraints of Problem Statement 7.3.2. Thus, we proceed as follows:

1. $(C2)$ $(Inv_{\mathcal{P}' \setminus X_n}) \subseteq Inv_{\mathcal{P}}$.

   By construction of $Inv_{\mathcal{P}'}$, $Inv_{\mathcal{P}'}$ is obtained by removing zero or more states in S. Therefore, this condition is trivially satisfied.

2. $(C3)$ $(T_{\mathcal{P}' \setminus X_n} | Inv_{\mathcal{P}' \setminus X_n}) \ \subseteq \ (T_{\mathcal{P}} | Inv_{\mathcal{P}' \setminus X_n})$.

   From the construction of $T_{\mathcal{P}'}$, the transitions in $T_{\mathcal{P}}$ are a subset of the transitions in $T_{\mathcal{P}}$. Note that all the recovery transitions that the algorithm Add_Nonmasking adds in Line 5 originate outside $Inv_{\mathcal{P}'}$. Therefore, condition $C3$ is met.

3. $(C4)$ $\mathcal{P}'$ is nonmasking F-tolerant to $SPEC$ from $Inv_{\mathcal{P}'}$.

   We distinguish two subgoals based on the behavior of $\mathcal{P}'$ in the absence and presence of faults:

   - First, we need to show that in the absence of faults, $\mathcal{P}' \models_{Inv_{\mathcal{P}'}} SPEC$. To this end, consider a computation $\overline{\sigma}$ of $T_{\mathcal{P}'}$ that starts in $Inv_{\mathcal{P}'}$: From 1, $\overline{\sigma}$ starts from a state in $Inv_{\mathcal{P}}$, and from 2, $\overline{\sigma}$ is a computation of $T_{\mathcal{P}}$. Moreover, since we remove deadlock states from $Inv_{\mathcal{P}'}$, if $\overline{\sigma}$ is infinite in $\mathcal{P}$ then it is infinite in $\mathcal{P}'$ as well. It follows that $\overline{\sigma} \in SPEC$. Thus, every computation of $T_{\mathcal{P}'}$ that starts from a state in S' is in $SPEC$. Also, by construction, $Inv_{\mathcal{P}'}$ is closed in $T_{\mathcal{P}'}$. Furthermore, since we remove deadlock regions from $Inv'^{r}_{\mathcal{P}}$, for all open regions, say r_0, in $Inv^r_{\mathcal{P}'}$ there exists an outgoing edge, say (r_0, r_1), for some $r_1 \in Inv^r_{\mathcal{P}'}$ where $r_0 \neq r_1$. Clearly, such an edge can only terminate at a different clock region which in turn advances time. This means that in the absence of faults our algorithm does not introduce time-convergent computations to $\mathcal{P}'$ and, hence, $\mathcal{P}' \models_{Inv_{\mathcal{P}'}} SPEC$.

- First, notice that by construction, $S_{\mathcal{P}'}$ is closed in $T_{\mathcal{P}}[]F$. Now, we need to show that every computation of $T_{\mathcal{P}'}[]F$ that starts from a state in $S_{\mathcal{P}'}$ reaches a state in $Inv_{\mathcal{P}'}$ within δ time units. Consider a computation $\overline{\sigma} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ of $T_{\mathcal{P}'}[]F$ that starts from a state in $S_{\mathcal{P}'}$. Let $r_0 \rightarrow r_1 \rightarrow \cdots$ be the corresponding sequence of regions. In the algorithm, after the loop terminates, by construction of G' (Phase 2) in the procedure Add_BoundedRecovery, the following holds:

$$\forall j : (((\mathrm{Rank}(\gamma(r_j)) \leq \delta \ \wedge \ (\mathrm{Rank}(\gamma(r_j)) > 0))) \Rightarrow \mathrm{Rank}(\gamma(r_{j+1})) < \mathrm{Rank}(\gamma(r_j))).$$

  Since the rank of regions are integers, it follows that there exists a region r_n such that $r_n \in Inv^r_{\mathcal{P}}$ which in turn means that for all states $\sigma \in r$ including the one that occurs in $\overline{\sigma}$, say σ_k at global time τ_k, $\sigma_k \in Inv_{\mathcal{P}'}$ and $\tau_k - \tau_0 \leq \delta$. Thus, computation $\overline{\sigma}$ reaches $Inv_{\mathcal{P}'}$ within δ. This also means that in the presence of faults as well, our algorithm does not introduce time-convergent computations to $\mathcal{P}'$, as it eventually reaches a state in the invariant.

Now, we show that the algorithm Add_Nonmasking is complete. Intuitively, in order to prove the completeness, we show that if our algorithm removes a state from $S_{\mathcal{P}}$ or $Inv_{\mathcal{P}}$ then no nonmasking program that meets the constraints of Problem Statement 7.3.2 can include that state in its fault-span or invariant. Thus, when our algorithm declares failure, it implies that there does not exist a legitimate state in fault-span or invariant of any solution to Problem Statement 7.3.2.

Observe that the algorithm removes a state, say σ_1, from $S_{\mathcal{P}} - Inv_{\mathcal{P}}$, if there does not exist a computation that starts from σ_1 and reaches a state in $Inv_{\mathcal{P}}$ within δ in the presence of faults. It follows that such a state cannot be present in the fault-span of any nonmasking program that satisfies the constraints of Problem Statement 7.3.2. Now, consider the case where σ_1 is reachable from another state, say σ_0, in $S_{\mathcal{P}}$ by taking fault transitions alone. It follows that σ_0 and all states in the corresponding computation from σ_0 to σ_1 cannot be present in the fault-span of any nonmasking program. Furthermore, if removal of such states create some deadlock states inside the program invariant then their removal is inevitable as well. Thus, states removed by our algorithm cannot be present in any nonmasking program.

Our algorithm declares failure when either the invariant or fault-span of the synthesized program is equal to the empty set. In other words, our algorithm fails to find a solution when all states of the intolerant program are illegitimate with respect to Problem Statement 7.3.2. Therefore, the algorithm Add_Nonasking is complete. ∎

Theorem 8.2.6 *The problem of adding nonmasking fault-tolerance to a real-time program is in PSPACE.*

Proof. Observe that the algorithm first generates the region graph of the intolerant program which is known to be in PSPACE [AD94]. Then, it performs a sequence of polynomial-time procedures (e.g., reachability analysis and finding shortest paths) in the size of the region graph. Therefore, our algorithm is in PSPACE. ∎

Case study (cont'd). Figure 8.6 illustrates the region graph generated by the Algorithm Add_Nonmasking given $\mathcal{AS}$ as the input program and recovery time $\delta = 1$. As can be seen, new recovery edges are added from all regions in the region fault-span to the regions in the invariant where at least one clock variable is reset or the regions where the system is in the STANDBY mode and no altitude sensor failure is reported. Note that, in the untimed version of $\mathcal{AS}$, a nonmasking program could have recovery transitions that originate from fault-span and end at any state in the invariant. However, in the timed setting, since we do not allow time decreases during recovery (cf. Line 5 in Algorithm 8.3), the algorithm merely adds recovery edges that either reset a clock variable or let it advance to a time-successor clock region. In the context of $\mathcal{AS}$, one may argue that recovery from fault-span to a state where the system is in AWAIT mode to power on the DOI is not desirable and recovery should be only possible to the INIT mode. Such a constraint can be specified by a safety property that needs to be met in the presence of faults. However, since nonmasking

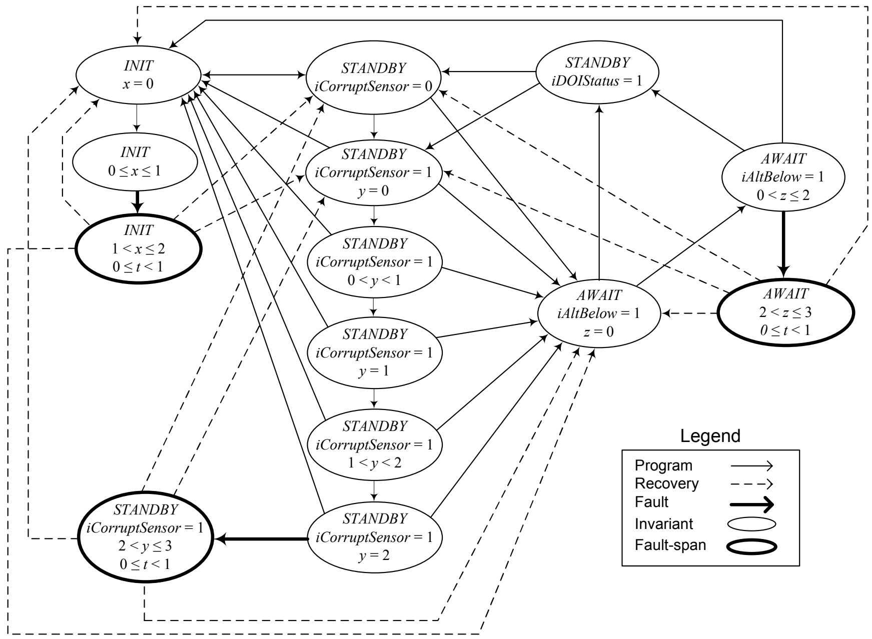


Figure 8.6: Region graph of nonmasking $\mathcal{AS}$.

$$
\begin{array}{llcl}
\mathcal{AS}9 :: & (0 \leq t < 1) & \xrightarrow{\{x\}//\{y,z\}} & mStatus := \textsc{Init} \\
\mathcal{AS}10 :: & (0 \leq t < 1) & \longrightarrow & mStatus, iCorruptSensor := \textsc{Standby}, 0 \\
\mathcal{AS}11 :: & (0 \leq t < 1) & \xrightarrow{\{y\}//\{x,z\}} & mStatus := \textsc{Standby} \\
\mathcal{AS}12 :: & (0 \leq t < 1) & \xrightarrow{\{z\}//\{x,y\}} & mStatus := \textsc{Await} \\
\mathcal{AS}13 :: & (0 \leq t < 1) & \longrightarrow & \mathbf{wait}
\end{array}
$$

Figure 8.7: Recovery timed guarded commands of nonmasking $\mathcal{AS}$.

programs are not required to satisfy safety properties in the presence of faults, nonmasking $\mathcal{AS}$ may contain such transitions. In Section 8.4, in synthesizing soft-masking $\mathcal{AS}$, we will constrain the recovery mechanism by a safety specification such that only desirable recovery transitions are added.

The algorithm does not remove any states from the invariant, as recovery is possible from any perturbed or delayed state (i.e., $ns = \{\}$). Hence, $Inv_{\mathcal{AS}}$. Recovery timed actions of nonmasking $\mathcal{AS}$ are presented in Figure 8.7. The rest of the timed guarded commands of nonmasking $\mathcal{AS}$ are the same as the intolerant $\mathcal{AS}$. The timed actions include transitions that are not illustrated in Figure 8.6. This is due to the reason that all the reachable regions are not shown in Figure 8.6. Timed actions associated with two sets of clock variables (e.g., $\{x\}//\{y, z\}$) means that the timed action resets all the clock variables in the first set (i.e.,

$\{x\}$) and a subset of the second set (i.e., either $\{\}$, $\{y\}$, $\{z\}$, or $\{y, z\}$).

8.3 Adding Soft and Hard-Failsafe Fault-Tolerance

In this section, we present our algorithms for synthesizing soft and hard-failsafe fault-tolerant programs. Recall that, intuitively, a soft-failsafe program is required to satisfy only the untimed part of its safety specification in the presence of faults. Also, recall that a hard-failsafe program satisfies its untimed part of safety specification as well as its timing constraints in the presence of fault.

8.3.1 Adding Soft-Failsafe Fault-Tolerance

As mentioned in Section 2.2, the safety specification consists of a set $SPEC_{bt}$ of bad transitions and a conjunction $SPEC_{\overline{br}}$ of multiple bounded response properties. In the presence of faults, a soft-failsafe program is required to maintain $SPEC_{\overline{bt}}$ only.

Algorithm sketch. We adapt the proposed algorithm in [KAE07], which adds failsafe fault-tolerance to *untimed* programs. Intuitively, our algorithm consists of three main phases. First, we identify and remove the set of states and transitions from where a sequence of faults alone may take the program to a state where a fault transition directly violates $SPEC_{\overline{bt}}$. In Phase 2, we ensure that this removal does not create new finite computations in the absence of faults. To this end, we remove deadlock states from the invariant. Finally, in Phase 3, we ensure the closure of the new program invariant in the set of program transitions.

We now describe the algorithm Add_SoftFalisafe (cf. Algorithm 8.6) in detail. We first transform the intolerant program $\mathcal{P} = \langle \Pi_\mathcal{P}, Inv_\mathcal{P} \rangle$ into its region graph $R(\mathcal{P}) = \langle \Pi^r_\mathcal{P}, Inv^r_\mathcal{P} \rangle$ (Line 1). Since the region graph is time-abstracted, we apply the algorithm for adding

---

Algorithm 8.6 Add_SoftFailsafe

Input: real-time program $\mathcal{P} = \langle \Pi_\mathcal{P}, Inv_\mathcal{P} \rangle$, fault transition predicate F, and timing independent safety specification represented by $SPEC_{bt}$

Output: a soft-failsafe fault-tolerant program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$

1: $\langle \Pi^r_\mathcal{P}, Inv^r_\mathcal{P} \rangle$, F^r, $SPEC^r_{bt}$:= ConstructRegionGraph($\langle \Pi_\mathcal{P}, Inv_\mathcal{P} \rangle$, F, $SPEC_{bt}$);
2: $T^r_{\mathcal{P}'}, Inv^r_{\mathcal{P}'}$:= Add_UntimedFailsafe($\langle \Pi^r_\mathcal{P}, Inv^r_\mathcal{P} \rangle, F^r, SPEC^r_{bt}$);
3: $\langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$:= ConstructRealTimeProgram($\langle \Pi^r_{\mathcal{P}'}, Inv^r_{\mathcal{P}'} \rangle$);
4: **return** $\langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$

---

failsafe fault-tolerance (from [KAE07]) to the region graph of $\mathcal{P}$ so that no edge in $SPEC^r_{bt}$ occurs in computations of $R(\mathcal{P})$ in the presence of faults. To this end, we invoke the procedure Add_UntimedFailsafe (Line 2).

The procedure Add_UntimedFailsafe (cf. Procedure 8.7) first identifies the set ms of regions from where $SPEC_{bt}$ may be violated by taking faults alone (Line 1) by calculating the smallest fixpoint of set of such regions. Next (Line 2), we compute the set mt of edges that consists of (1) edges that directly violate safety (i.e., transitions in $SPEC_{bt}$) or (2) edges whose target region is in ms (i.e., edges that lead a computation to a state from where safety may be violated by faults alone). Since programs do not have control over occurrence of faults, we remove the set ms (respectively, the set mt) from the region invariant $Inv^r_{\mathcal{P}}$ (respectively, set of program edges $T^r_{\mathcal{P}}$) of $R(\mathcal{P})$ (Line 3). Then, we remove deadlock regions from $Inv^r_{\mathcal{P}}$ (Line 4) to ensure that removal of ms from $Inv^r_{\mathcal{P}}$ does not introduce new finite computations to $R(\mathcal{P}) = \langle \Pi^r_{\mathcal{P}}, Inv^r_{\mathcal{P}} \rangle$. Since we may remove a set of regions from $Inv^r_{\mathcal{P}}$, we ensure the closure of $Inv^r_{\mathcal{P}}$ in $T^r_{\mathcal{P}}$ (Line 9). Finally, we return the failsafe region graph $R(\mathcal{P}') = \langle \Pi^r_{\mathcal{P}'}, Inv^r_{\mathcal{P}'} \rangle$ (Line 10).

After computing the failsafe region graph $R(\mathcal{P}') = \langle \Pi^r_{\mathcal{P}'}, Inv^r_{\mathcal{P}'} \rangle$ in the algorithm Add_SoftFailsafe (Line 2), we transform the $R(\mathcal{P}')$ back into a real-time program $\mathcal{P}'$ (Line 3).

Theorem 8.3.1 *The algorithm Add_SoftFalisafe is sound and complete.*

Proof. First, we show that the algorithm is sound. Similar to the proof of Theorem 8.2.5,

---

Procedure 8.7 Add_UntimedFailsafe

Input: region graph $\langle \Pi^r_{\mathcal{P}}, Inv^r_{\mathcal{P}} \rangle$, set of faults edges F^r, and set of edges $SPEC^r_{bt}$
Output: an untimed failsafe fault-tolerant program $\langle \Pi^r_{\mathcal{P}'}, Inv^r_{\mathcal{P}'} \rangle$

1: $ms := \{r_0 \mid \exists r_1, r_2 \cdots r_n :$
   $(\forall j \mid 0 \leq j < n : (r_j, r_{j+1}) \in f^r) \ \wedge \ (r_{n-1}, r_n) \in SPEC^r_{bt} \ \}$; {Phase 1}
2: $mt := \{(r_0, r_1) \mid r_1 \in ms \ \vee \ (r_0, r_1) \in SPEC^r_{bt} \ \}$;
3: $Inv^r_{\mathcal{P}}, \ T^r_{\mathcal{P}} := Inv^r_{\mathcal{P}} - ms, \ T^r_{\mathcal{P}} - mt$; {Phase 2}
4: $Inv^r_{\mathcal{P}} :=$ RemoveDeadlocks($Inv^r_{\mathcal{P}}, T^r_{\mathcal{P}}$);
5: **if** $(Inv^r_{\mathcal{P}} = \{\})$ **then**
6:   `declare no soft/hard-failsafe` F`-tolerant program` $\mathcal{P}'$ `exists`;
7:   **exit**();
8: **end if**
9: $T^r_{\mathcal{P}} := T^r_{\mathcal{P}} - \{(r_0, r_1) \mid r_0 \in Inv^r_{\mathcal{P}} \ \wedge \ r_1 \notin Inv^r_{\mathcal{P}}\}$; {Phase 3}
10: **return** $T^r_{\mathcal{P}}, Inv^r_{\mathcal{P}}$;

---

we proceed as follows:

1. $(C2)$ $(Inv_{\mathcal{P}'\backslash X_n}) \subseteq Inv_{\mathcal{P}}$.

   By construction, $Inv_{\mathcal{P}'}$ is obtained by removing zero or more states in $Inv_{\mathcal{P}}$. Therefore, this condition is trivially satisfied.

   $(C3)$ $(T_{\mathcal{P}'\backslash X_n}|Inv_{\mathcal{P}'\backslash X_n}) \subseteq (T_{\mathcal{P}}|Inv_{\mathcal{P}'\backslash X_n})$.
   From the construction of $T_{\mathcal{P}}$, the transitions in $T_{\mathcal{P}}$ are a subset of the transitions in $T_{\mathcal{P}}$. Therefore, condition $C2$ is met.

2. $\mathcal{P}'$ is soft-failsafe F-tolerant to $SPEC$ from $Inv_{\mathcal{P}'}$.

   We need to show two subgoals based on the behavior of $\mathcal{P}'$ in the absence and presence of faults. Proof of $\mathcal{P}' \models_{Inv_{\mathcal{P}'}} SPEC$ is exactly the same as the proof of the first subgoal of Theorem 8.2.5. Now, we show that $\mathcal{P}'$ never violates $SPEC_{\overline{bt}}$ in the presence of F. Let MS (respectively, MT) be the state predicate (respectively, set of transitions) that corresponds to ms (respectively, mt). We let the fault-span $S_{\mathcal{P}'}$ to be the set of states reached in any computation of $T_{\mathcal{P}'}[]F$ that starts from a state in $Inv_{\mathcal{P}'}$. Consider a computation prefix $\overline{\alpha}$ of $T_{\mathcal{P}'}[]F$ that starts from a state in $S_{\mathcal{P}'}$. From the definition of $S_{\mathcal{P}'}$, there exists a computation prefix $\overline{\alpha}'$ of $T_{\mathcal{P}'}[]F$ such that $\overline{\alpha}$ is a suffix of $\overline{\alpha}'$ and $\overline{\alpha}'$ starts from a state in $Inv_{\mathcal{P}'}$. If $\overline{\alpha}'$ violates $SPEC_{\overline{bt}}$ then there exists a prefix of $\overline{\alpha}'$, say $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots (\sigma_n, \tau_n)$, such that $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots (\sigma_n, \tau_n)$ violates $SPEC_{\overline{bt}}$. Without loss of generality, let $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots (\sigma_n, \tau_n)$ be the smallest such prefix. It follows that (σ_{n-1}, σ_n) violates $SPEC_{\overline{bt}}$ and, hence, $(\sigma_{n-1}, \sigma_n) \in MT$. By construction, $T_{\mathcal{P}'}$ does not contain any transitions in MT. Thus, (σ_{n-1}, σ_n) is a transition of F . If (σ_{n-1}, σ_n) is a transition of F then $\sigma_{n-1} \in MS$. It follows that $(\sigma_{n-2}, \sigma_{n-1}) \in MT$. By the same argument, $(\sigma_{n-2}, \sigma_{n-1})$ is a transition of f. Hence, $\sigma_{n-2} \in MS$. Continuing thus, by induction, if $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots (\sigma_n, \tau_n)$ violates $SPEC_{\overline{bt}}$, $\sigma_0 \in MS$, which is a contradiction since $\sigma_0 \in S'$. Thus, each prefix of $\overline{\alpha}'$ maintains $SPEC_{\overline{bt}}$ and, hence, $T_{\mathcal{P}'}[]F$ maintains $SPEC_{\overline{bt}}$ from $S_{\mathcal{P}'}$. Note that in the presence of faults a soft-failsafe program need not to recover to its invariant. In other words, in the presence of faults, a soft-failsafe program is allowed to deadlock or violate its liveness specification. Hence, we are not required to show that a soft-failsafe program does not exhibit time-convergent computations in the presence of faults.

Now, we show that the algorithm Add_SoftFailsafe is complete. Let program $\mathcal{P}'' = \langle \Pi_{\mathcal{P}''}, Inv_{\mathcal{P}''} \rangle$ solve the Problem Statement 8.2.4. Clearly, $Inv^r_{\mathcal{P}''} \cap ms = \{\}$; if $r_0 \in Inv^r_{\mathcal{P}''} \cap ms$ then the execution of faults alone from r_0 takes the program to a state from where $SPEC_{\overline{bt}}$ may be violated. It follows that $Inv^r_{\mathcal{P}''} \subseteq (Inv^r_{\mathcal{P}} - ms)$. Likewise, $T^r_{\mathcal{P}''}|Inv^r_{\mathcal{P}''}$ cannot include any edges in mt. Hence, $T^r_{\mathcal{P}''} \subseteq (T^r_{\mathcal{P}} - mt)$. Finally, every computation of $\mathcal{P}''$ that starts from a state in $Inv_{\mathcal{P}''}$ must be infinite if it were to be in $SPEC$. It follows that there exists a nonempty subset of $Inv^r_{\mathcal{P}}$, namely $Inv^r_{\mathcal{P}''}$, such that all computations of $T^r_{\mathcal{P}} - mt$ within that subset are infinite.

Our algorithm declares that no solution for the synthesis problem exists only when there is no nonempty subset of $Inv^r_{\mathcal{P}} - ms$ such that all the computations of $T^r_{\mathcal{P}} - mt$ within that subset are infinite. It follows that our algorithm declares failure only if there exists no soft-failsafe fault-tolerant program that solves Problem Statement 7.3.2. ∎

Remark 8.3.2 We note that, the completeness proof of Theorem 8.3.1 implies that the outcome of adding soft-failsafe is a *maximal* program in the sense that every state that is removed by Add_SoftFailsafe has to be removed, i.e., such states cannot be present in any soft-failsafe program that satisfies the constraints of Problem Statement 7.3.2.

Theorem 8.3.3 *The problem of adding soft-failsafe fault-tolerance to a real-time program is in PSPACE.*

Proof. Observe that our algorithm first generates the region graph of the intolerant program which is known to be in PSPACE [AD94]. Then it performs a sequence of polynomial time procedures (mainly reachability analysis) in the size of region graph. Therefore, our algorithm is in PSPACE. ∎

8.3.2 Adding Hard-Failsafe Fault-Tolerance with One Bounded Response Property

In this subsection, we present our algorithm Add_HardFailsafe for the case where the synthesized hard-failsafe program is required to satisfy only one bounded response property in the presence of faults, i.e., $SPEC_{\overline{br}} \equiv P \mapsto_{\leq \delta} Q$.

Algorithm sketch. Intuitively, the algorithm works in three phases. First, we add soft-failsafe fault-tolerance to $\mathcal{P}$ to ensure that transitions in $SPEC_{bt}$ occur in no computation of $\mathcal{P}'$. In Phase 2, similar to addition of nonmasking, we recompute the set of program transitions, invariant, and fault-span in a loop until we reach a fixpoint. In this loop, the goal is to remove computations of $T_{\mathcal{P}}[]F$ that violate $SPEC_{\overline{br}} \equiv P \mapsto_{\leq \delta} Q$. In Phase 3, we resolve a special case where a state in $Inv_{\mathcal{P}} \cap Q$ becomes a deadlock state. Note that since such a state is in the invariant, we need to ensure that its removal will not create new deadlock states unnecessarily. Thus, If such a deadlock state exists in $Inv_{\mathcal{P}} \cap Q$, we remove it from both $Inv_{\mathcal{P}}$ and Q and rerun the algorithm.

We now describe the algorithm in detail (cf. Algorithm 8.8):

- (*Phase 1*) In order to ensure that $\mathcal{P}'$ maintains $SPEC_{\overline{bt}}$, we first add soft-failsafe fault-tolerance to $\mathcal{P}$ (Line 1). We then transform $\mathcal{P}$ into its region graph $R(\mathcal{P})$ (Line 2). Our first estimate of invariant of $\mathcal{P}'$ is the invariant generated by Add_SoftFailsafe and our first estimate of fault-span of $\mathcal{P}'$ is obtained by removing the regions in ms (as calculated in Add_SoftFailsafe) from the region space.

Procedure 8.8 Add_HardFailsafe

Input: real-time program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$, fault transition predicate F, timing independent safety specification represented by $SPEC_{bt}$, bounded response property $P \mapsto_{\leq \delta} Q$, and **integer** n
Output: a hard-failsafe fault-tolerant program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$

1: $\langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle :=$ Add_SoftFailsafe$(\langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle, F, SPEC_{bt})$; {Phase 1}
2: $\langle \Pi^r_{\mathcal{P}}, Inv^r_{\mathcal{P}} \rangle, P^r, Q^r, F^r :=$ ConstructRegionGraph$(\langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle, P, Q, F)$;
3: $S^r_1, Inv^r_1 := \mathcal{S}^r_{\mathcal{P}} - ms, Inv^r_{\mathcal{P}}$;
4: **repeat** {Phase 2}
5:    $S^r_2, Inv^r_2 := S^r_1, Inv^r_1$;
6:    $T^r_{\mathcal{P}_1} := T^r_{\mathcal{P}} \mid Inv^r_1 \ \cup \ \{((s_0, \rho_0), (s_1, \rho_1)) \mid (s_0, \rho_0) \in (S^r_1 - Inv^r_1) \ \wedge \ (s_1, \rho_1) \in S^r_1 \ \wedge$
        $\exists \rho_2 \mid \rho_2$ is a time-successor of $\rho_0 \ : \ (\exists \lambda \subseteq X \ : \ \rho_1 = \rho_2[\lambda - \{t\} := 0])\} - mt$;
7:    $T^r_{\mathcal{P}_1}, ns :=$ Add_BoundedRecovery$(\langle \Pi^r_{\mathcal{P}_1}, Inv^r_{\mathcal{P}_1} \rangle, F^r, P^r, Q^r, n, \delta)$;
8:    $S^r_1 :=$ ConstructFaultSpan$(S^r_1 - ns, \ F^r)$;
9:    $Inv^r_1 :=$ RemoveDeadlocks$(Inv^r_1 \ \cap \ S^r_1, T^r_{\mathcal{P}_1})$
10:    $qrm := Q^r \cap (Inv^r_{\mathcal{P}} - Inv^r_1)$; {Phase 3}
11:    **if** $(qrm \neq \{\})$ **then**
12:       $Inv^r_{\mathcal{P}} := Inv^r_{\mathcal{P}} - qrm$;
13:       $T^r_{\mathcal{P}} := T^r_{\mathcal{P}} - \{(r_0, r_1) \mid (r_0 \in qrm) \ \vee \ (r_1 \in qrm)\}$;
14:       $Q^r := Q^r - qrm$;
15:       **goto** Line 3;
16:    **end if**
17:    **if** $(Inv^r_1 = \{\} \ \vee \ S^r_1 = \{\})$ **then**
18:       `declare no hard-failsafe` f`-tolerant program` $\mathcal{P}'$ `exists`;
19:       **exit**();
20:    **end if**
21: **until** $(S^r_1 = S^r_2 \ \wedge \ Inv^r_1 = Inv^r_2)$;
22: $\langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle :=$ ConstructRealTimeProgram$(\langle \Pi^r_{\mathcal{P}_1}, Inv^r_{\mathcal{P}_1} \rangle$

- (*Phase 2*) We now modify $R(\mathcal{P})$, such that any computation that starts from a region in P^r, reaches a region in Q^r within δ time units even in the presence of faults. We achieve this in the same fashion as we added bounded-time recovery for synthesizing nonmasking programs in Section 8.2; we recompute the set $T^r_{\mathcal{P}_1}$ of edges, region fault-span S^r_1, and region invariant Inv^r_1 until we reach a fixpoint. However, there exist two differences: (1) we exclude the set mt from $T^r_{\mathcal{P}_1}$ to ensure that the program does not violate $SPEC_{\overline{bt}}$ from fault-span (Line 6), and (2) we invoke the procedure Add_BoundedRecovery for region predicates P^r and Q^r (Line 7).

- (*Phase 3*) Since the procedure Add_BoundedRecovery does not include all legitimate computations that start in P and end in Q, we need to consider a special case where a region, say r_1, in $Inv^r_1 \cap Q^r$ becomes a deadlock region (Line 9). In this case, it is possible that all the regions along a path that starts from some region, say r_0, in P^r and end in r_1 become deadlock regions. Hence, our algorithm needs to identify a new path from r_0 to a region in Q^r other than r_1. Thus, in such a case, we remove such deadlock regions and associated edges from $Inv^r_{\mathcal{P}} \cap Q^r$ and $T^r_{\mathcal{P}_1}$ (Line 13-15) and rerun the algorithm with the updated region predicates (Line 15).

  Finally, the algorithm transforms the synthesized region graph $R(\mathcal{P}_1) = \langle \Pi^r_{\mathcal{P}_1}, Inv^r_{\mathcal{P}_1} \rangle$ back into a real-time program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$ (Line 22).

Theorem 8.3.4 *The algorithm Add_HardFalisafe is sound and complete.*

Proof. Again, in order to prove the soundness, we show that our algorithm meets the constraints of Problem Statement 7.3.2. We only show that the output of our algorithm is hard-failsafe F-tolerant to $SPEC$ from $Inv_{\mathcal{P}'}$, as the rest of the proof is the same as proof of theorems 8.2.5 and 8.3.1.

In order to prove that $\mathcal{P}'$ is hard-failsafe F-tolerant to $SPEC$ from $S_{\mathcal{P}'}$, we need to show that (1) $\mathcal{P}'[]F$ maintains $SPEC_{\overline{bt}}$ from $S_{\mathcal{P}'}$, and (2) $\mathcal{P}'[]F$ maintains $SPEC_{\overline{br}} = P \mapsto_{\leq \delta} Q$ from $S_{\mathcal{P}'}$. Since the algorithm Add_HardFailsafe first adds soft-failsafe fault-tolerance to $\mathcal{P}$ (Line 1) and the edges that are added in Line 6 are disjoint from mt, the first subgoal of the proof is immediately discharged by applying Theorem 8.3.1.

Let $\overline{\alpha}$ be a computation prefix of $T_{\mathcal{P}'}[]F$ that starts from a state in $S_{\mathcal{P}'}$. From the definition of $S_{\mathcal{P}'}$, there exists a computation prefix $\overline{\alpha}'$ of $T_{\mathcal{P}'}[]F$ such that $\overline{\alpha}$ is a suffix of $\overline{\alpha}'$ and $\overline{\alpha}'$ starts from a state in $Inv_{\mathcal{P}'}$. If $\overline{\alpha}'$ violates $SPEC_{\overline{br}}$ then there exists a prefix of

$\overline{\alpha}'$, say $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots (\sigma_n, \tau_n)$, such that $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots (\sigma_n, \tau_n)$ violates $SPEC_{\overline{br}}$. Without loss of generality, let $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots (\sigma_n, \tau_n)$ be the smallest such prefix. It follows that there exists $i \geq 0$ such that (1) $\sigma_i \in P$, (2) $\tau_n - \tau_i > \delta$, and (3) for all k, $i \leq k \leq n$, $\sigma_k \notin Q$. Let r be the region that contains σ_i. It follows that in the corresponding MaxDelay digraph $\text{Rank}(\gamma(r)) > \delta$, which implies that $r \in ns$. In this case, (σ_{i-1}, σ_i) is not in $T_{\mathcal{P}}$. Thus, (σ_{i-1}, σ_i) is a transition of f. If (σ_{i-1}, σ_i) is a transition of f then the procedure ConstructFaultSpan identifies σ_{i-1} and removes it from $S_{\mathcal{P}}$, which in turn removes the incident transitions in the next iteration (Line 6). By the same argument, $(\sigma_{i-2}, \sigma_{i-1})$ is a transition of F and, hence, σ_{n-2} should not be present in $S_{\mathcal{P}'}$ as well. Continuing thus, by induction, if $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots (\sigma_n, \tau_n)$ violates $SPEC_{\overline{br}}$, σ_0 is explored by the procedure ConstructFaultSpan, which is a contradiction, as we initially assumed that $\sigma_0 \in Inv_{\mathcal{P}'}$. Thus, each prefix of $\overline{\alpha}'$ maintains $SPEC_{\overline{br}}$. It follows that $T_{\mathcal{P}'}[]F$ maintains $SPEC_{\overline{br}}$ from $S_{\mathcal{P}'}$. Note that in the presence of faults a hard-failsafe program need not to recover to its invariant. In other words, in the presence of faults, a hard-failsafe program is allowed to deadlock or violate its liveness specification. Hence, we are not required to show that a hard-failsafe program does not exhibit time-convergent computations in the presence of faults.

We now show that the algorithm is complete. Intuitively, we show that any state that is removed by our algorithm cannot be present in any hard-failsafe program. First, observe that in Theorem 8.3.1, we showed that the outcome of adding soft-failsafe fault-tolerance is a maximal program in the sense that any state removed by the algorithm Add_SoftFailsafe has to be removed (cf. Remark 8.3.2). Hence, in order to prove the completeness of the algorithm Add_HardFailsafe, we only need to focus on the case where the algorithm fails to synthesize $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$ such that $T_{\mathcal{P}'}[]F$ maintains $SPEC_{\overline{br}}$ from $S_{\mathcal{P}'}$. Observe that the algorithm removes a state, say σ_1, in P, if there does not exist a computation that starts from σ_1 and reaches a state in Q within δ in the presence of faults. Notice that such a state cannot be present in any hard-failsafe program. Moreover, if σ_1 is reachable from another state, say σ_0, in $S_{\mathcal{P}'}$ by taking fault transitions alone then σ_0 cannot be present in the fault-span of any hard-failsafe program as well. Furthermore, if removal of states such as σ_0 and σ_1 creates deadlock states inside the program invariant then their removal is also inevitable (since $\mathcal{P}'$ must meet constraint $C3$ of Problem Statement 7.3.2). Notice that Phase 2 of the algorithm ensures that nonmaximality of the set of transitions does not induce deadlock states unnecessarily. Hence, states removed by our algorithm are present in no hard-failsafe program. Our algorithm declares failure when the invariant of the synthesized program is equal to the empty set. In other words, our algorithm fails to find a solution when all states of the intolerant program are illegitimate with respect to Problem Statement 7.3.2. Therefore, the algorithm Add_HardFailsafe is complete. ∎

Complexity of adding hard-failsafe fault-tolerance. We now show that the problem of adding hard-failsafe fault-tolerance to real-time programs, where the synthesized program is required to satisfy only one bounded response property, is PSPACE-complete. Our reduction is from the reachability problem in timed automata known to be PSPACE-complete [AD94].

Instance. A real-time program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$, a set of faults F, and safety specification $SPEC$ such that $\mathcal{P} \models_{Inv_{\mathcal{P}}} SPEC$.

Hard-failsafe synthesis decision problem. Does there exist a real-time program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$ such that $\mathcal{P}'$ is hard-masking F-tolerant to $SPEC$ from $Inv_{\mathcal{P}'}$?

Theorem 8.3.5 *The hard-failsafe synthesis decision problem is PSPACE-complete in the size of the input intolerant program.*

Proof. Membership to PSPACE follows immediately from the space complexity of the algorithm

Add_HardFailsafe. Now, we show that the problem is PSPACE-hard. To this end, we reduce the reachability problem in timed automata to the above decision problem.

Mapping. Let $\mathcal{A} = \langle L, L^0, X, I, E \rangle$ be an instance of the reachability problem in timed automata. Let s_0 and s_1 be two locations in L. Without loss of generality, we assume that $s_0 \in L^0$. We map $\mathcal{A}$ to an instance of our problem as follows:

- (*State space*) The state space of $\mathcal{P}$ consists of all legitimate states of $\mathcal{A}$. That is,

$$\mathcal{S}_{\mathcal{P}} = \{(s, \nu) \mid (s \in L) \ \wedge \ (\nu \models I(s))\}.$$

- (*Program transitions*) The set of program transitions of $\mathcal{P}$ is obtained by removing all transitions in $\mathcal{A}$ that originate from s_1 and then by adding a self-loop at location s_1. That is,

$$T_{\mathcal{P}} = \{(s_1,\nu) \to (s_1,\nu) \mid \nu \models I(s_1)\} \;\cup\; \{(s,\nu) \to (t,\mu) \mid (s \neq s_1) \;\wedge$$
$$\exists\langle s',\lambda,\varphi,t'\rangle \in E : (s'=s) \;\wedge\; (t'=t) \;\wedge\; (\nu \models \varphi) \;\wedge\; (\mu \models I(t'))\}.$$

- (*Invariant*) The invariant of $\mathcal{P}$ consists of states whose location is s_1:

$$Inv_{\mathcal{P}} = \{(s_1,\nu) \mid \nu \models I(s_1)\}.$$

- (*Fault transitions*) We let the set of fault transitions to be $F = \{(s_1,\nu) \to (s_0, \nu[X := 0]) \mid \nu \models I(s_1)\}$. Note that in timed automata, it is assumed that all clock variables are reset in initial locations.

- (*Safety specification*) Let $SPEC_{bt}$ consist of all transitions that are *not* present in $\mathcal{A}$. Furthermore, let $SPEC_{\overline{br}} \equiv P \mapsto_{\leq \delta} Q$, where:

  - $P = \{(s_0,\nu) \mid \nu \models I(s_0)\}$,
  - $Q = \{(s_1,\nu) \mid \nu \models I(s_1)\}$, and
  - We choose the recovery time to be unbounded, i.e., $\delta = \infty$.

Reduction. If s_1 is reachable from s_0 in $\mathcal{A}$ then there exists a computation in $\mathcal{A}$ which starts from a state whose location is s_0 and ends at a state whose location is s_1. A real-time program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'}\rangle$ constructed from this computation plus the self-loop at s_1 is clearly a hard-failsafe program with invariant $Inv_{\mathcal{P}'} = Inv_{\mathcal{P}}$ and meets the constraints of Problem Statement 7.3.2. Now, we show the other direction. Let us assume that the answer to the decision problem is affirmative, i.e., we can synthesize a hard-failsafe program $\mathcal{P}'$ with invariant $Inv_{\mathcal{P}'}$ by starting from $\mathcal{P}$. If this is the case then state predicate Q must be reachable from state predicate P in $\mathcal{P}'$. Note that due to the way we constructed $SPEC_{bt}$, we ruled out hard-failsafe programs that satisfy $SPEC_{\overline{br}}$ by adding recovery transitions outside the invariant $Inv_{\mathcal{P}'}$. Thus, in $\mathcal{P}'$, a computation that starts from P reaches Q using transitions in $\mathcal{A}$ alone. Hence, location s_1 must be reachable from location s_0 in timed automaton $\mathcal{A}$. ∎

Case study (cont'd). We now demonstrate that how our algorithm synthesizes a hard-failsafe version of $\mathcal{AS}$ with respect to $SPEC_{bt}$ and $SPEC_{\overline{br_3}}$ as defined in case study of Subsection 2.2. Figures 8.8 and 8.9 illustrate the region graph and revised timed guarded commands of hard-failsafe $\mathcal{AS}$ respectively. Observe that the timed action $\mathcal{AS}3$ in the

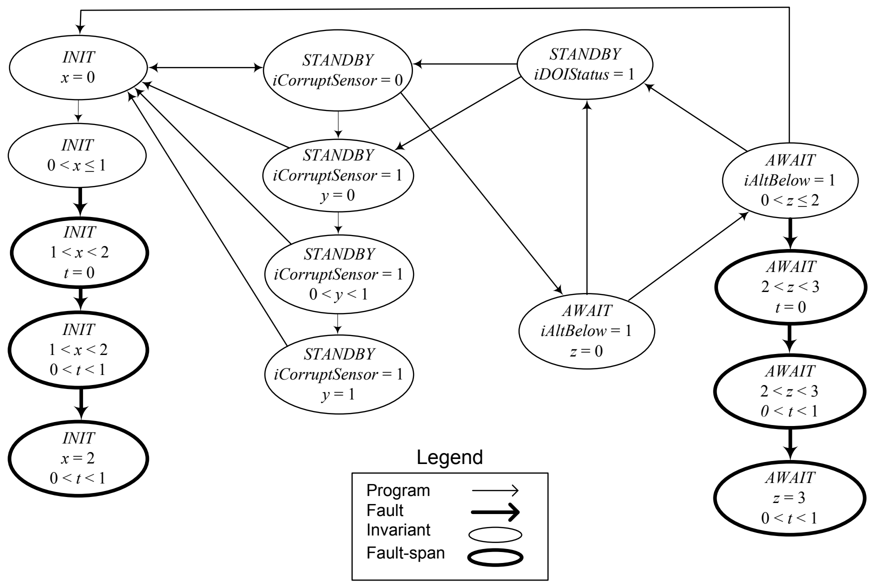


Figure 8.8: Region graph of hard-failsafe $\mathcal{AS}$.

intolerant program does not maintain $SPEC_{\overline{bt}}$ when the altitude meter shows an invalid value due to occurrence of fault action $F4$. It is easy to see that when the algorithm Add_SoftFailsafe is invoked in Line 2:

- $ms = \{\}$, and
- $mt = SPEC^r_{bt}$.

Thus, invocation of the algorithm Add_SoftFailsafe removes edges that originate from Standby regions, where $iCorruptSensor = 1$, and end at Await regions. In terms of timed guarded commands, the algorithm strengthens the guard of timed action $\mathcal{AS}3$ such that $iCorruptSensor \neq 1$.

Additionally, $SPEC_{\overline{br_3}}$ requires that if the program fails to read the altitude meter, it should initialize within 2 seconds even in the presence of the delay fault action $F2$. It is easy to observe that in the algorithm Add_HardFailsafe: $ns = \{(s, \rho) \mid 1 < y(\rho) \leq 3\}$. Thus, the algorithm removes edges that advance clock variable y beyond $y = 1$. In terms of timed guarded commands, the algorithm strengthens delay action $\mathcal{AS}6$ such that $y \leq 1$. In fact, the hard-failsafe $\mathcal{AS}$ tolerates $F2$ by taking delay transitions for at most 1 second while in the Standby mode. The invariant of hard-failsafe $\mathcal{AS}$ is as follows:

$$Inv_{\text{HFS\_ASW}} = \{\sigma \mid \ ((mStatus(\sigma) = \text{INIT}) \ \Rightarrow \ (x(\sigma) \leq 1)) \ \wedge$$

$$
\begin{aligned}
&((mStatus(\sigma) = \textsc{Standby}) \;\Rightarrow \\
&\qquad (iCorruptSensor(\sigma) = 0 \;\vee\; \mathbf{y}(\sigma) \leq \mathbf{1})) \wedge \\
&((mStatus(\sigma) = \textsc{Await}) \;\Rightarrow\; (z(\sigma) \leq 2))\}.
\end{aligned}
$$

$$
\begin{array}{lll}
\textsc{Hfs\_}\mathcal{AS}3 :: (mStatus = \textsc{Standby}) \wedge (iAltBelow = 0) \wedge (\mathbf{iCorruptSensor} \neq \mathbf{1}) \wedge & & \\
\qquad (iDOIStatus = 0) & \xrightarrow{\{z\}} & mStatus, iAltBelow := \textsc{Await}, 1 \\
\textsc{Hfs\_}\mathcal{AS}6 :: ((mStatus = \textsc{Standby}) \wedge (iCorruptSensor = 0)) \vee & & \\
\qquad (x \leq 1) \vee (\mathbf{y} \leq \mathbf{1}) \vee (z \leq 2) & \longrightarrow & \mathbf{wait}
\end{array}
$$

Figure 8.9: Revised timed guarded commands of hard-failsafe $\mathcal{AS}$.

8.4 Adding Soft and Hard-Masking Fault-Tolerance

In this section, we present our results on automated synthesis of soft and hard-masking fault-tolerant programs. Recall that, intuitively, a soft-masking program is required to satisfy only the untimed part of its safety specification and provides bounded-time recovery in the presence of faults. Also, recall that a hard-masking program satisfies its untimed part of safety specification as well as its timing constraints and provides bounded-time recovery in the presence of fault.

8.4.1 Adding Soft-Masking Fault-Tolerance

In order to synthesize a soft-masking program, we should generate a program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$ and fault-span $S_{\mathcal{P}'}$, such that $\mathcal{P}'[]F$ maintains $SPEC_{\overline{bt}}$ and if faults perturb the state of a program to a state in $S_{\mathcal{P}'}$, it recovers to $Inv_{\mathcal{P}'}$ within recovery time δ. Since the algorithm Add_SoftMasking (cf. Algorithm 8.9) is very similar to addition of nonmasking fault-tolerance, we only present the algorithm sketch. In fact, the only difference between the two algorithms is in soft-masking, in addition to bounded-time recovery from fault-span to the invariant, the program also guarantees that $SPEC_{\overline{bt}}$ is never violated. We also do not present a proof of soundness and completeness, as combining proofs of theorems 8.3.1 and 8.2.5 trivially shows soundness and completeness of Add_SoftMasking.

Algorithm sketch. The algorithm consists of two main phases. In Phase 1, our first estimate of a soft-masking program is a soft-failsafe program. Then, in Phase 2, we

Algorithm 8.9 Add_SoftMasking

Input: real-time program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$, transition predicate F, **integers** n, δ
Output: a soft-masking fault-tolerant program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$ with bounded-time recovery δ.

$\langle \Pi^r_{\mathcal{P}}, Inv^r_{\mathcal{P}} \rangle, F^r :=$ ConstructRegionGraph($\langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle, F$); {Phase 1}
$S^r_1 := \mathcal{S}^r_{\mathcal{P}}$;
repeat {Phase 2}
  $S^r_2, Inv^r_2 \ := \ S^r_1, Inv^r_1$;
  $T^r_{\mathcal{P}_1} := T^r_{\mathcal{P}}|Inv^r_1 \ \cup \ \{((s_0, \rho_0), (s_1, \rho_1)) \mid (s_0, \rho_0) \in (S^r_1 - Inv^r_1) \ \wedge \ (s_1, \rho_1) \in S^r_1 \ \wedge \ \exists \rho_2 \mid \rho_2$ is a time-successor of $\rho_0 \ : \ (\exists \lambda \subseteq X \ : \ \rho_1 = \rho_2[\lambda - \{t\} := 0])\} - mt$;
  $T^r_{\mathcal{P}_1}, ns :=$ Add_BoundedRecovery($\langle \Pi^r_{\mathcal{P}_1}, Inv^r_{\mathcal{P}_1} \rangle, F^r, S^r_1 - Inv^r_1, Inv^r_1, n, \delta$);
  $S^r_1 :=$ ConstructFaultSpan($S^r_1 - ns, \ F^r$);
  $Inv^r_1 :=$ RemoveDeadlocks($Inv^r_1 \ \cap \ S^r_1, T^r_{\mathcal{P}_1}$);
  if ($Inv^r_1 = \{\} \ \vee \ S^r_1 = \{\}$) **then**
    `declare no nonmasking` F`-tolerant program` $\mathcal{P}'$ `exists;`
    exit();
  end if
until ($S^r_1 = S^r_2 \ \wedge \ Inv^r_1 = Inv^r_2$);
$\langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle :=$ ConstructRealTimeProgram($\langle \Pi^r_{\mathcal{P}_1}, Inv^r_{\mathcal{P}_1} \rangle$); {Phase 3}

recompute the set of program transitions, invariant, and fault-span so that the fault-span is closed in $T_{\mathcal{P}}[]f$, and from each state in the fault-span, there exists a path which ends at a state in the invariant within δ and maintains $SPEC_{\overline{bt}}$.

Theorem 8.4.1 *The algorithm* Add_SoftMasking *is sound and complete.* ∎

Theorem 8.4.2 *The problem of adding soft-masking fault-tolerance to a real-time program is PSPACE-complete in the size of the input intolerant program.*

Proof. Membership to PSPACE follows immediately from the space complexity of the algorithm Add_SoftMasking. Now, we show that the problem is PSPACE-hard. To this end, we reduce the reachability problem in timed automata to the soft-masking synthesis decision problem. Mapping the elements of reachability problem in timed automata to an instance of soft-masking synthesis decision problem is the same as our mapping in Theorem 8.3.5 except the following changes:

- (*Safety specification*) Let $SPEC_{bt}$ consist of all transitions that are *not* present in $\mathcal{A}$. Furthermore, we let $SPEC_{\overline{br}}$ be any arbitrary conjunction of bounded response properties.

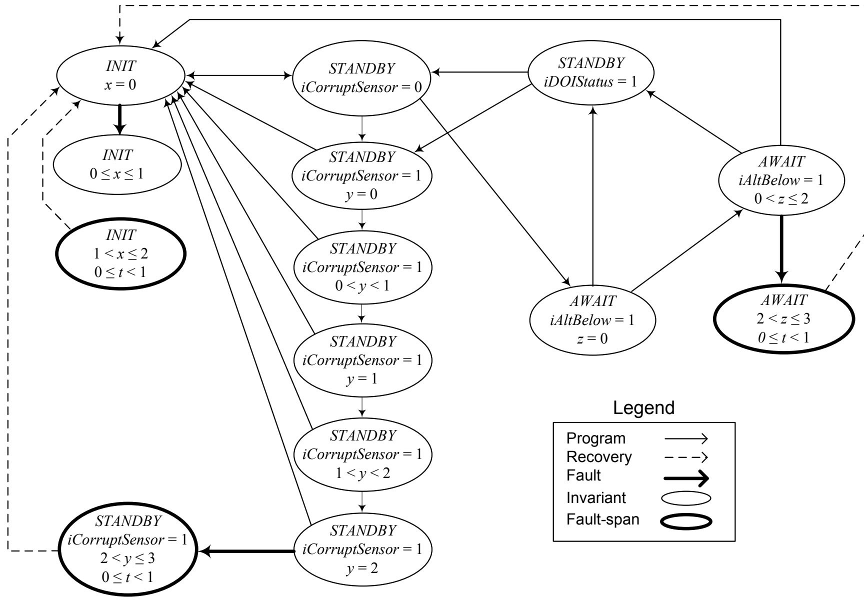


Figure 8.10: Region graph of soft-masking $\mathcal{AS}$.

- (*Recovery time*) We choose the recovery time from fault-span to the invariant to be unbounded, i.e., $\delta = \infty$. This means that even adding soft-masking fault-tolerance to real-time programs with unbounded recovery time is PSPACE-hard.

Let $S_{\mathcal{P}'} = \{(s_0, \nu) \mid \nu \models I(s_0)\}$. If s_1 is reachable from s_0 in $\mathcal{A}$ then there exists a computation in $\mathcal{A}$ which starts from a state whose location is s_0 and ends at a state whose location is s_1. A real-time program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$ constructed from this computation plus the self-loop at s_1 is clearly a soft-masking program with invariant $Inv_{\mathcal{P}'} = Inv_{\mathcal{P}}$. Now, let us assume that we can synthesize a soft-masking program $\mathcal{P}'$ with invariant $Inv_{\mathcal{P}'}$ by starting from $\mathcal{P}$. If this is the case then $Inv_{\mathcal{P}'}$, where $Inv_{\mathcal{P}'} \subseteq Inv_{\mathcal{P}}$, must be reachable from $S_{\mathcal{P}'}$ in $\mathcal{P}'$. Note that due to the way we constructed $SPEC_{bt}$, we ruled out soft-masking programs that provide recovery from their fault-span to $Inv_{\mathcal{P}'}$ by adding recovery transitions outside $Inv_{\mathcal{P}'}$. Thus, in $\mathcal{P}'$, a computation that starts from $S_{\mathcal{P}'}$ reaches $Inv_{\mathcal{P}'}$ using transitions in $\mathcal{A}$ alone. Hence, location s_1 must be reachable from location s_0 in timed automaton $\mathcal{A}$. ∎

Case study (cont'd). We now demonstrate that how our algorithm synthesizes a soft-masking version of $\mathcal{AS}$ with respect to $SPEC_{bt}$ and recovery time $\delta = 1$. Figures 8.10 and 8.11 illustrate the region graph and recovery and revised timed guarded commands of soft-masking $\mathcal{AS}$, respectively. It is easy to observe that synthesizing soft-masking $\mathcal{AS}$

$$\begin{array}{llll}
\textsc{Smk_}\mathcal{AS}3 :: & (\mathit{mStatus} = \textsc{Standby}) \wedge (\mathit{iAltBelow} = 0) \wedge (\mathit{iCorruptSensor} \neq 1) \wedge & & \\
& (\mathit{iDOIStatus} = 0) & \xrightarrow{\{z\}} & \mathit{mStatus}, \mathit{iAltBelow} := \textsc{Await}, 1 \\
\textsc{Smk_}\mathcal{AS}9 :: & (0 \leq t < 1) & \xrightarrow{\{x\}//\{y,z\}} & \mathit{mStatus} := \textsc{Init} \\
\textsc{Smk_}\mathcal{AS}10 :: & (0 \leq t < 1) & \longrightarrow & \textbf{wait}
\end{array}$$

Figure 8.11: Recovery and revised timed guarded commands of soft-masking $\mathcal{AS}$.

involves strengthening the program invariant as we did in adding hard-failsafe, plus adding recovery transitions as we did for adding nonmasking fault-tolerance. However, according to $SPEC_{bt}$, recovery transitions are constrained such that only recovery to INIT mode is possible. The rest of the timed guarded commands remain unchanged. Finally, the invariant of the soft-masking program is the same as the invariant of the intolerant program, i.e, $Inv_{smk\_\mathcal{AS}} = Inv_{\mathcal{AS}}$.

8.4.2 Adding Hard-Masking Fault-Tolerance

As mentioned in Section 7.1, a hard-masking program satisfies both $SPEC_{\overline{bt}}$ and $SPEC_{\overline{br}}$ and provides bounded-time recovery in the presence of faults. Let us consider the case where $SPEC_{\overline{br}}$ consists of only two bounded response properties. In this subsection, we show that synthesizing a hard-masking program in this setting is NP-complete in the size of the program's region graph. Note, however, that the complexity of this problem remains open for the case where $SPEC_{\overline{br}}$ is defined as a single bounded response property.

Instance. A real-time program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$, a set of faults f, and safety specification $SPEC$, where $SPEC_{\overline{br}} \equiv (P \mapsto_{\leq \delta} Q) \wedge (R \mapsto_{\leq \theta} U)$, and $\mathcal{P} \models_{Inv_{\mathcal{P}}} SPEC$.

Hard-masking synthesis decision problem. Does there exist a program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$ such that $\mathcal{P}'$ is obtained from the above instance by adding hard-masking fault-tolerance and satisfies the constraints of Problem Statement 7.3.2?

Theorem 8.4.3 *Hard-masking synthesis decision problem is NP-complete in the size of the region graph.*

Proof. Clearly, having a solution to the problem, we can verify it in polynomial time. Hence, the problem is in NP. In order to show that the problem is NP-hard, we adapt the NP-hardness proof of the problem of adding two unbounded response properties to an untimed programs in the absence of faults [EKB05]. We prove this theorem by a reduction from the 3-SAT problem. The 3-SAT problem is as follows: Let $x_1, x_2 \cdots x_n$ be propositional variables. Given is a Boolean formula $y = y_1 \ \wedge \ y_2 \ \cdots \ y_M$, where each y_j, $1 \leq j \leq M$, is a disjunction of exactly three literals. The decision problem determines whether there exists an assignment of truth values to $x_1, x_2 \cdots x_n$ such that y is satisfiable.

Mapping. Mapping of propositional variables and disjunctions to an instance of our decision problem is as follows:

- (*State space, invariant, and state predicates* P, Q, R, *and* U) For each variable x_i in the given 3-SAT instance, we introduce six states P_i, a_i, Q_i, R_i, b_i, and U_i, where $1 \leq i \leq n$ (cf. Figure 8.12). For each disjunction y_j, we introduce a state t_j, where $1 \leq j \leq M$. We also introduce the state s, which represents the program invariant. We let the set of clock variables of $\mathcal{P}$ be the empty set. Put it another way, in our instance, states of $\mathcal{P}$ are not associated with clock valuations. Thus, we have:

  - $\mathcal{S}_\mathcal{P} = \{P_i, a_i, Q_i, R_i, b_i, U_i \mid 1 \leq i \leq n\} \ \cup \ \{t_j \mid 1 \leq j \leq M\} \ \cup \ \{s\}$,
  - $Inv_\mathcal{P} = \{s\}$, and
  - $P = \{P_i \mid 1 \leq i \leq n\}$, $Q = \{Q_i \mid 1 \leq i \leq n\}$, $R = \{R_i \mid 1 \leq i \leq n\}$, and $U = \{U_i \mid 1 \leq i \leq n\}$.

- (*Program transitions*) For each variable x_i, we include $(P_i, a_i), (a_i, b_i)$, (b_i, Q_i), (Q_i, s), $(R_i, b_i), (b_i, a_i)$, (a_i, U_i), and (U_i, s) in the set $T_\mathcal{P}$ of program transitions (cf.

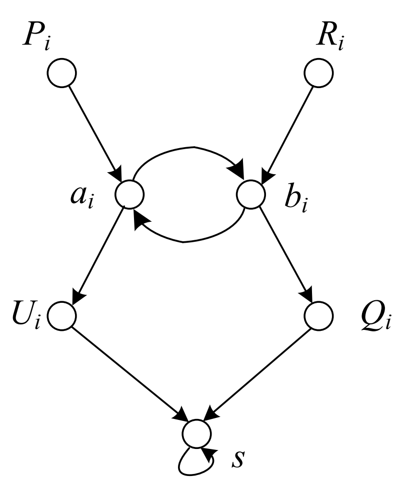


Figure 8.12: Mapping 3-SAT to addition of hard-masking fault-tolerance.

Figure 8.12). We also add a self-loop at state s. Moreover, corresponding to each disjunction y_j, we include the following transitions:

- If x_i is a literal in y_j then we include the transition (t_j, P_i).
- If $\neg x_i$ is a literal in y_j then we include the transition (t_j, R_i).

- (*Fault transitions*) We also add fault transitions from s to all states that correspond to a disjunction. That is, $F = \{(s, t_j) \mid 1 \leq j \leq M\}$.

- (*Safety specification*) We let the set of bad transitions to be the empty set, i.e., $SPEC_{bt} = \{\}$. Moreover, in $SPEC_{\overline{br}}$, we let the response times to be unbounded, i.e., $\delta = \theta = \infty$.

Reduction. Now, we show that the given instance of 3-SAT is satisfiable iff we are able to synthesize $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$ by adding hard-masking fault-tolerance to $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$ (identified above) such that $\mathcal{P}'$ meets the constraints of Problem Statement 7.3.2. We distinguish two subgoals:

1. ($\Rightarrow$) First, we show that if the given instance of the 3-SAT formula is satisfiable then there exists a hard-masking program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$ derived from $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$. Since the 3-SAT formula is satisfiable, there exists an assignment of truth values to variables x_i, $1 \leq i \leq n$, such that each disjunction y_j, $1 \leq j \leq M$, becomes *true*. We construct $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$ as follows.

   - The invariant of $\mathcal{P}'$ is identical to the invariant of $\mathcal{P}$, i.e., $Inv_{\mathcal{P}'} = Inv_{\mathcal{P}}$.
   - For each variable x_i, if x_i is *true* then we include the transitions (P_i, a_i), (a_i, b_i), (b_i, Q_i), and (Q_i, s) in $T_{\mathcal{P}'}$.
   - For each variable x_i, if x_i is *false* then we include the transitions (R_i, b_i), (b_i, a_i), (a_i, U_i), and (U_i, s) in $T_{\mathcal{P}'}$.
   - For each disjunction y_j that contains x_i, we include the transition (t_j, P_i) in $T_{\mathcal{P}'}$ if x_i is *true*.
   - For each disjunction y_j that contains $\neg x_i$, we include the transition (t_j, R_i) in $T_{\mathcal{P}'}$ if x_i is *false*.
   - Finally, we let the self-loop (s, s) to be present in $T_{\mathcal{P}'}$ as well.

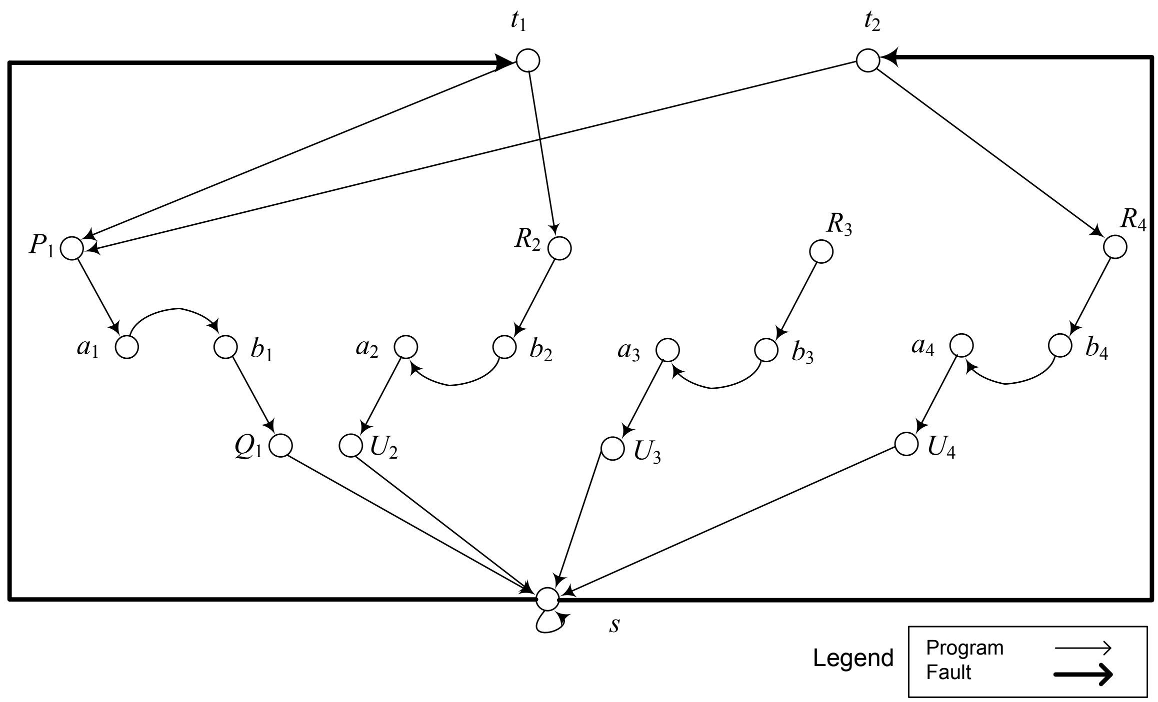


Figure 8.13: Partial structure of the hard-masking program.

As an illustration, we show the partial structure of $\mathcal{P}'$, for the formula $[(x_1 \vee \neg x_2 \vee x_3) \ \wedge \ (x_1 \vee x_2 \vee \neg x_4)]$, where $x_1 = \mathit{true}$, $x_2 = \mathit{false}$, $x_3 = \mathit{false}$, and $x_4 = \mathit{false}$ in Figure 8.13. Now, we show that if we construct $\mathcal{P}'$ as prescribed above, $\mathcal{P}'$ is hard-masking F-tolerant to SPEC from $\mathit{Inv}_{\mathcal{P}'}$:

- Clearly, the first two constraints of the Problem Statement 7.3.2 are satisfied.
- It is easy to observe that by construction, there are no deadlock states. Moreover, if the state of $\mathcal{P}'$ is perturbed by a fault transition, the corresponding computation reaches P_i (i.e., $x_i = \mathit{true}$) and then that computation will eventually reach Q_i and finally recovers back to s. This is due to the fact that $T_{\mathcal{P}'}$ does not include the transition (b_i, a_i). Likewise, if a computation of $T_{\mathcal{P}'}[]f$ reaches R_i (i.e., $x_i = \mathit{false}$) then that computation will eventually reach U_i and recovers back to s. Again, this is because $T_{\mathcal{P}'}$ does not include the transition (a_i, b_i). Thus, $\mathcal{P}'$ is hard-masking f-tolerant to SPEC from $\mathit{Inv}_{\mathcal{P}'}$.

2. ($\Leftarrow$) Next, we show that if there exists a hard-masking program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, \mathit{Inv}_{\mathcal{P}'} \rangle$ such that $\mathcal{P}'$ meets the constraints of Problem Statement 7.3.2 then the corresponding 3-SAT formula is satisfiable. We assign truth values to the propositional variables as

follows. If there exists a computation of $T_{\mathcal{P}'}[]f$, such that state P_i is reachable then we assign x_i the truth value *true*. Otherwise, we assign the truth value *false*.

We now show that the above truth assignment satisfies all disjunctions. Observe that since S' is nonempty then $Inv_{\mathcal{P}'} = \{s\}$. Let y_j be any disjunction and let t_j be the corresponding state in $\mathcal{P}'$. Since t_j is a state in the fault-span and $\mathcal{P}'$ is a hard-masking program, $\mathcal{P}'$ cannot deadlock, i.e., there must be some outgoing transition from t_j. This transition terminates in either P_i or R_i, for some i. If the transition from t_j terminates in P_i then y_j contains literal x_i and x_i is assigned the truth value *true*. Hence, y_j evaluates to true. If the transition from t_j terminates in R_i then P_i is unreachable. Otherwise, (i) transitions $(R_i, b_i), (b_i, a_i)$, and (a_i, U_i) must be included to ensure that $R \mapsto_{\leq\infty} U$ is satisfied, and (ii) transitions $(P_i, a_i), (a_i, b_i)$, and (b_i, Q_i) must also be included to guarantee that $P \mapsto_{\leq\infty} Q$ is satisfied. Since the inclusion of all six transitions (P_i, a_i), (a_i, b_i), (b_i, Q_i), (R_i, b_i), (b_i, a_i), and (a_i, U_i) creates a cycle and, hence, causes violation of both $P \mapsto_{\leq\infty} Q$ and $R \mapsto_{\leq\infty} U$, it follows that P_i must be unreachable in any computation of $T_{\mathcal{P}'}[]f$ if R_i is reachable. Thus, if R_i is reachable then x_i will be assigned the truth value *false*. Since in this case y_j contains $\neg x_i$, the disjunction y_j evaluates to *true*. Therefore, the assignment of values considered above is a satisfying truth assignment for the corresponding 3-SAT formula. ∎

Chapter 9

Synthesizing Bounded-Time Phased Recovery

In this chapter, we focus on the problem of automated synthesis for real-time systems that provide bounded-time phased recovery in the presence of faults. To motivate the notion of phased recovery, we return to our traffic controller example (see Section 2.1.2).

Consider the case where sig_0 is green and sig_1 is red. If the timer that is responsible for changing sig_1 from red to green is reset due to a circuit problem, sig_1 may turn green within some time while sig_0 is also green. Obviously, this is a violation of the safety specification. In order to transform this system into a fault-tolerant one, it is desirable to synthesize a version of the original system, in which even in the presence of faults, the system

1. never executes a transition in $SPEC_{bt_{\mathcal{TC}}}$, and

2. always meets the following bounded-time recovery specification denoted by $SPEC_{\overline{br}_{\mathcal{TC}}}$: when the system state is in $\neg Inv_{\mathcal{TC}}$, it must reach a state in $Inv_{\mathcal{TC}}$ within a bounded amount of time.

Although such a recovery mechanism is necessary in a fault-tolerant real-time system, it may not be sufficient. In particular, one may require that the system must initially reach a special set of states, say Q, within some time θ, and subsequently recover to $Inv_{\mathcal{TC}}$ within δ time units. We call the set Q an *intermediate recovery predicate*. The intuition for such *phased recovery* comes from the requirement that the occurrence of faults must be noted (e.g., for scheduling hardware repairs or replacement) *before* normal system operation resumes. Thus, in our example, Q could be the set of states where all signals are red. Such a constraint ensures that the system first goes to a state in which a set of preconditions for final recovery (e.g., via a system reboot or rollback) is fulfilled.

The main results in this chapter are as follows:

- We formally define the notion of bounded-time phased recovery in the context of fault-tolerant real-time systems.
- We show that, in general (i.e., when $Q \not\subseteq \mathit{Inv}_{\mathcal{P}}$ and $\mathit{Inv}_{\mathcal{P}} \not\subseteq Q$), the problem of synthesizing fault-tolerant real-time programs that provide phased recovery is NP-complete. An example of such a case is the traffic signals system in which Q includes states where all signals are flashing red. This result also shows that the problem of adding hard-masking fault-tolerance with only one timing constraint is also NP-complete. This result generalizes Theorem 8.4.3.
- We characterize a sufficient condition for cases where the synthesis problem can be solved efficiently. In particular, we show that if $\mathit{Inv}_{\mathcal{P}} \subseteq Q$, and, execution of the synthesized system needs to be *closed* in Q (i.e., starting from a state in Q, the state of the system never leaves Q) then there exists a polynomial-time sound and complete synthesis algorithm in the size of the input intolerant program's region graph. An example of such a case is the traffic signals system in which Q is the set of states where either both signals remain red indefinitely or $\mathit{Inv}_{\mathcal{TC}}$ holds.

9.1 Bounded-Time Phased Recovery

As mentioned earlier, preserving safety specification and providing simple recovery to the invariant from the fault-span may not be sufficient and, hence, it may be necessary to complete recovery to the invariant in a sequence of phases where each phase satisfies certain constraints. We formalize the notion of bounded-time phased recovery by a set of bounded response properties inside the safety specification, i.e., by SPEC_{br} (cf. Definition 2.2.4). In this paper, in particular, we focus on 2-*phase recovery.*

Definition 9.1.1 (2-phase recovery) Let $\mathcal{P} = \langle \Pi_{\mathcal{P}}, \mathit{Inv}_{\mathcal{P}} \rangle$ be a real-time program, Q be an arbitrary *intermediate recovery predicate*, F be a set of faults, and SPEC be a specification (as defined in Definitions 2.2.4 and 2.2.5). We say that $\mathcal{P}$ *provides* 2-phase recovery from $Inv_{\mathcal{P}}$ and Q with recovery times $\delta, \theta \in \mathbb{Z}_{\geq 0}$, respectively, iff $\mathcal{P}[]F$ maintains $SPEC_{\overline{br}}$ from $Inv_{\mathcal{P}}$, where $SPEC_{\overline{br}} \equiv (\neg Inv_{\mathcal{P}} \mapsto_{\leq \theta} Q) \;\wedge\; (Q \mapsto_{\leq \delta} Inv_{\mathcal{P}})$. ∎

Note that in Definition 9.1.1, if $Inv_{\mathcal{P}}$ and Q are disjoint then $\mathcal{P}$ has to recover to Q and then $Inv_{\mathcal{P}}$ in order, as $Inv_{\mathcal{P}}$ is closed in $\mathcal{P}$. On the other hand, if $Inv_{\mathcal{P}}$ and Q are not disjoint, $\mathcal{P}$ has the following options: (1) recover to $Q \cap \neg Inv_{\mathcal{P}}$ within θ and then $Inv_{\mathcal{P}}$, or (2) directly recover to $Inv_{\mathcal{P}} \cap Q$ within $\min(\delta, \theta)$.

Example (cont'd). As described earlier in this Section, when faults F_0 or F_1 (defined in Subsection 7.2.3) occur, the program $\mathcal{TC}$ has to, first, ensure that nothing catastrophic happens and then recover to its normal behavior. Thus, the fault-tolerant version of $\mathcal{TC}$ has to, first, reach a state where both signals remain red indefinitely and subsequently recover to $Inv_{\mathcal{P}}$ where exactly one signal turns green. In particular, we let the 2-phase recovery specification of $\mathcal{TC}$ be the following:

$$SPEC_{\overline{br}_{\mathcal{TC}}} \equiv (\neg Inv_{\mathcal{TC}} \mapsto_{\leq 3} Q_{\mathcal{TC}}) \;\wedge\; (Q_{\mathcal{TC}} \mapsto_{\leq 7} Inv_{\mathcal{TC}}),$$

where $Q_{\mathcal{TC}} = \forall i \in \{0, 1\} \;:\; (sig_i = R) \;\wedge\; (z_i > 1)$. The response times in $SPEC_{\overline{br}_{\mathcal{TC}}}$ (i.e., 3 and 7) are simply two arbitrary numbers to express the duration of the two phases of recovery.

9.2 Complexity of Synthesizing Bounded-Time 2-Phase Recovery

In this section, we show that, in general, the problem of synthesizing fault-tolerant real-time programs that provide phased recovery is NP-complete in the size of locations of the given fault-intolerant real-time program.

Instance. A real-time program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$, a set of faults F, and a specification $SPEC$, such that $\mathcal{P} \models_{Inv_{\mathcal{P}}} SPEC$, where $SPEC_{\overline{br}} \equiv (\neg Inv_{\mathcal{P}} \mapsto_{\leq \theta} Q) \;\wedge\; (Q \mapsto_{\leq \delta} Inv_{\mathcal{P}})$ for state predicate Q and $\delta, \theta \in \mathbb{Z}_{\geq 0}$.

The decision problem (FTPR). Does there exist an F-tolerant program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$ such that $\mathcal{P}'$ meet the constraints of Problem Statement 7.3.2?

We now show that the FTPR problem is NP-complete to decide. To this end, we reduce a variation of the *2-path problem* [FHW80] to FTPR.

The simplified 2-path problem (2PP). Given are a digraph $G = \langle V, A \rangle$ and three distinct vertices $v_1, v_2, v_3 \in V$. Decide whether G has a simple (v_1, v_3)-path that also contains the vertex v_2 [BJG02].

Theorem 9.2.1 *The problem of transforming a fault-intolerant real-time program into a fault-tolerant program that provides bounded-time phased recovery is NP-complete in the size of locations of the fault-intolerant program.*

Proof.

Since showing membership to NP is straightforward, we only show that the problem is NP-hard.

Mapping. Given an instance of the 2PP problem (i.e., $G = \langle V, A \rangle$, v_1, v_2, and v_3), we first present a polynomial-time mapping from the 2PP instance to an instance of the FTPR problem (i.e., $\mathcal{P} = \langle \Pi_\mathcal{P}, Inv_\mathcal{P} \rangle$, S, f, and $SPEC$) as follows (see also Figure 9.1):

- (*clock variables*) $X = \{\}$,
- (*state space*) $\mathcal{S}_\mathcal{P} = \{s_v \mid v \in V\}$,
- (*program transitions*) $T_\mathcal{P} = (\{(s_u, s_v) \mid (u, v) \in A\} \cup \{(s_{v_3}, s_{v_3})\}) - \{(s_{v_3}, s_u) \mid u \in V - \{v_3\}\}$,
- (*invariant*) $Inv_\mathcal{P} = \{s_{v_3}\}$,
- (*fault transitions*) $F = \{(s_{v_3}, s_{v_1})\}$,
- (*safety specification*) $SPEC_{bt} = \{\}$ and $SPEC_{\overline{br}} \equiv (\neg S \mapsto_{\leq \theta} Q) \wedge (Q \mapsto_{\leq \delta} S)$, where $Q = \{s_{v_2}\}$, $\delta = \infty$, and $\theta = \infty$.

An intuitive description of the above mapping is as follows. First, we let the set of clock variables of $\mathcal{P}$ be the empty set. This is simply because our goal is to show that the problem is NP-hard in the size of *locations* of the input program. The state space of $\mathcal{P}$

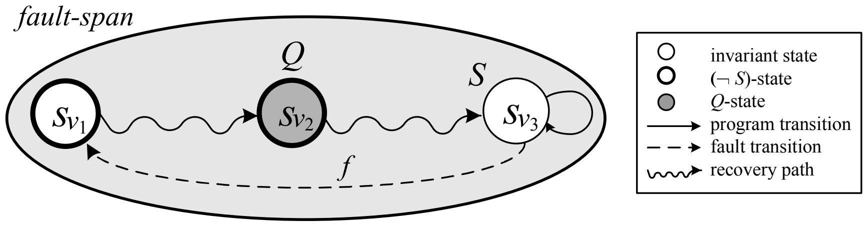


Figure 9.1: Mapping 2-path problem to fault-tolerance synthesis.

consists of all vertices in V. The program invariant $Inv_{\mathcal{P}}$ merely includes state s_{v_3}. The set of program transitions consists of all arcs in A except arcs that originate at v_3, plus a self-loop at s_{v_3}. Inclusion of the self-loop guarantees that all program computations are infinite. Exclusion of arcs that originate from v_3 ensures the closure of $Inv_{\mathcal{P}}$. There is no bound on recovery times. Finally, we let $SPEC_{bt}$ be the empty set so that all program transitions can potentially participate in the synthesized program.

Reduction. Given the above mapping, we now show that 2PP has a solution iff the answer to the FTPR problem is affirmative:

- ($\Rightarrow$) Let the answer to the 2PP be a simple path π that originates at v_1, ends at v_3, and contains v_2. We claim that in the structure shown in Figure 9.1, the set of program transitions $T_{\mathcal{P}'}$ obtained by taking only the transitions corresponding to the arcs along π plus the self-loop (s_{v_3}, s_{v_3}) satisfies the constraints of Problem Statement 7.3.2. We prove our claim as follows. Notice that (1) $\mathcal{S}_{\mathcal{P}} = \mathcal{S}_{\mathcal{P}'}$, (2) $Inv_{\mathcal{P}'} = Inv_{\mathcal{P}} = \{s_{v_3}\}$, (3) $T_{\mathcal{P}'}|Inv_{\mathcal{P}'} \subseteq T_{\mathcal{P}}|Inv_{\mathcal{P}'}$, and (4) $\mathcal{P}'$ is fault-tolerant to $SPEC$ from $Inv_{\mathcal{P}'}$, as (i) in the absence of faults, by starting from the invariant $Inv_{\mathcal{P}'}$, all computations of $T_{\mathcal{P}'}$ are infinite, and (ii) in the presence of faults, $\mathcal{P}' \models_{Inv_{\mathcal{P}'}} SPEC_{\overline{br}}$ (since π is a simple path that meets Q and S, respectively), and, $\mathcal{P}' \models_{Inv_{\mathcal{P}'}} SPEC_{\overline{bt}}$.
- ($\Leftarrow$) Let the answer to the FTPR problem be $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$ with invariant $Inv_{\mathcal{P}'}$. Since $Inv_{\mathcal{P}'}$ must be nonempty, $Inv_{\mathcal{P}'} = \{s_{v_3}\}$. Now, consider a computation prefix of $\mathcal{P}'$ that starts from $Inv_{\mathcal{P}'}$ and the fault transition (s_{v_3}, s_{v_1}) perturbs the state of $\mathcal{P}'$. Since $\mathcal{P}'$ is fault-tolerant it must satisfy the bounded response properties $\neg Inv_{\mathcal{P}'} \mapsto_{\leq\theta} Q$ and $Q \mapsto_{\leq\delta} Inv_{\mathcal{P}'}$. Hence, there should exist a computation prefix $\overline{\sigma}$ that originates at $\{s_{v_1}\}$ and reaches $Q = \{s_{v_2}\}$. Moreover, $\overline{\sigma}$ must also visit $Inv_{\mathcal{P}'} = \{s_{v_3}\}$. Notice that starting from s_{v_1}, $\overline{\sigma}$ must first visit s_{v_2} and subsequently s_{v_3}. This is because if $\overline{\sigma}$

first visits s_{v_3} then due to the closure of $Inv_{\mathcal{P}'}$ in $T_{\mathcal{P}'}$, it will never reach s_{v_2}. Now, we claim that a path, say π, whose vertices and arcs correspond to state and transitions in $\overline{\sigma}$, is the answer to 2PP. We prove this claim as follows. Obviously, π starts from v_1, ends at v_3 and contains v_2. Moreover, since $\overline{\sigma}$ reaches both $Inv_{\mathcal{P}'}$ and Q in a finite number of steps, $\overline{\sigma}$ cannot contain cycles. Therefore, π is a simple path. ∎

Example (cont'd). The proof of Theorem 9.2.1 particularly implies that if Q and $Inv_{\mathcal{P}}$ are disjoint in the problem instance then NP-completeness of the synthesis problem is certain. In the context of $\mathcal{TC}$, notice that according to the definitions of $Inv_{\mathcal{TC}}$ and $Q_{\mathcal{TC}}$ in Sections 2.2 and 9.1, it is the case that $Inv_{\mathcal{TC}} \cap Q_{\mathcal{TC}} = \{\}$. Hence, the $\mathcal{TC}$ program and specification in their current form exhibit an instance where the synthesis problem is NP-complete. However, in Subsection 9.3.1, we demonstrate that a slight modification in the specification of $\mathcal{TC}$ makes the problem significantly easier to solve. ∎

9.3 A Sufficient Condition for a Polynomial-Time Solution

In this section, we present a sufficient condition under which one can devise a polynomial-time sound and complete solution to the Problem Statement 7.3.2 in the size of time-abstract bisimulation of input program.

Claim 9.3.1 *Let $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$ be a program and recovery specification $SPEC_{\overline{br}} \equiv (\neg Inv_{\mathcal{P}} \mapsto_{\leq \theta} Q) \;\wedge\; (Q \mapsto_{\leq \delta} Inv_{\mathcal{P}})$. There exists a polynomial-time sound and complete solution to Problem Statement 7.3.2 in the size of the region graph of $\mathcal{P}$, if $(Inv_{\mathcal{P}} \subseteq Q) \wedge (Q$ is closed in $T_{\mathcal{P}'})$.* ∎

In order to validate this claim, we propose the Algorithm Add_BoundedPhasedRecovery.

Algorithm sketch. Intuitively, the algorithm works as follows. In Step 1, we transform the input program into a region graph. In Step 2, we isolate the set of states from where $SPEC_{\overline{bt}}$ may be violated. In Step 3, we ensure that any computation of $\mathcal{P}'$ that starts from a state in $\neg Inv_{\mathcal{P}'} - Q$ (respectively, $Q - Inv_{\mathcal{P}'}$) reaches a state in Q (respectively, $Inv_{\mathcal{P}'}$) within θ (respectively, δ) time units. In Step 4, we ensure the closure of fault-span and deadlock freedom of invariant. We repeat Steps 3-4 until a fixpoint is reached. Finally, in Step 5, we transform the resultant region graph back into a real-time program.

Algorithm 9.1 Add_BoundedPhasedRecovery

Input: A real-time program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$, fault transitions F, bad transitions $SPEC_{bt}$, intermediate recovery predicate Q s.t. $Inv_{\mathcal{P}} \subseteq Q$, recovery time δ, and intermediate recovery time θ.

Output: If successful, a fault-tolerant real-time program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$.

$\langle \Pi^r_{\mathcal{P}}, Inv^r_{\mathcal{P}} \rangle$, Q^r, f^r, $SPEC^r_{bt}$:= ConstructRegionGraph($\langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$, Q, f, $SPEC_{bt}$);
$ms \ := \ \{r_0 \mid \exists r_1, r_2 \cdots r_n \ : \ (\forall j \mid 0 \leq j < n : (r_j, r_{j+1}) \in F^r) \quad \wedge \quad (r_{n-1}, r_n) \in SPEC^r_{bt}\}$;
$mt \ := \ \{(r_0, r_1) \mid (r_1 \in ms) \ \vee \ ((r_0, r_1) \in SPEC^r_{bt})\}$;
$S^r_1 \ := \ \mathcal{S}^r_{\mathcal{P}} - ms$;
repeat
  $S^r_2, \ Inv^r_2 \ := \ S^r_1, \ Inv^r_1$;
  $T^r_{\mathcal{P}_1} \ := \ T^r_{\mathcal{P}} | Inv^r_1 \ \cup \ \{((s_0, \rho_0), (s_1, \rho_1)) \mid (s_0, \rho_0) \in (S^r_1 - Q^r) \ \wedge \ (s_1, \rho_1) \in T^r_1 \ \wedge$
    $\exists \rho_2 \mid \rho_2$ is a time-successor of $\rho_0 \ : \ (\exists \lambda \subseteq X \ : \ \rho_1 = \rho_2[\lambda := 0])\} \ \cup$
      $\{((s_0, \rho_0), (s_1, \rho_1)) \mid (s_0, \rho_0) \in (Q^r - Inv^r_1) \ \wedge \ (s_1, \rho_1) \in Q^r \ \wedge$
    $\exists \rho_2 \mid \rho_2$ is a time-successor of $\rho_0 \ : \ (\exists \lambda \subseteq X \ : \ \rho_1 = \rho_2[\lambda := 0])\} \ - \ mt$;
  $T^r_{\mathcal{P}_1}, \ ns \ :=$ Add_rtUNITY($\langle \Pi^r_{\mathcal{P}_1}, Inv^r_{\mathcal{P}_1} \rangle$, $S^r_1 - Q^r$, Q^r, θ);
  $S^r_1 \ := \ S^r_1 \ - \ ns$;
  $T^r_{\mathcal{P}_1}, \ ns \ :=$ Add_rtUNITY($\langle \Pi^r_{\mathcal{P}_1}, Inv^r_{\mathcal{P}_1} \rangle$, $Q^r - Inv^r_1$, Inv^r_1, δ);
  $S^r_1, \ Q^r \ := \ S^r_1 \ - \ ns, \ Q^r \ - \ ns$;
  while $(\exists r_0, r_1 : r_0 \in S^r_1 \ \wedge \ r_1 \notin S^r_1 \ \wedge \ (r_0, r_1) \in F^r)$ **do**
    $S^r_1 := S^r_1 - \{r_0\}$;
  end while
  while $(\exists r_0 \in (Inv^r_1 \cap S^r_1) \ : \ (\forall r_1 \mid (r_1 \neq r_0 \ \wedge \ r_1 \in Inv^r_1) \ : \ (r_0, r_1) \ \notin \ T^r_{)_1})$ **do**
    $Inv^r_1 := Inv^r_1 - \{r_0\}$;
  end while
  if $(Inv^r_1 = \{\} \ \vee \ S^r_1 = \{\})$ **then**
    `‘‘declare no fault-tolerant program exists’’`; **exit;**
  end if
until $(S_1 = S_2 \ \wedge \ Inv_1 = Inv_2)$
$\langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle \ :=$ ConstructRealTimeProgram($\langle \Pi^r_{\mathcal{P}_1}, Inv^r_{\mathcal{P}_1} \rangle$);
return $\langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$;

Assumption 1 Let $\overline{\alpha} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots (\sigma_n, \tau_n)$ be a computation prefix where $\sigma_0, \sigma_n \in S$ and $\sigma_i \notin S$ for all $i \in \{1..n-1\}$. Only for simplicity of presentation, we assume that the number of occurrence of faults in $\overline{\alpha}$ is one. Precisely, we assume that in $\overline{\alpha}$, only (σ_0, σ_1) is a fault transition and no faults occur outside the program invariant. In Chapter 8, we have shown how to deal with cases where multiple faults occur in a computation when adding bounded response properties. The same technique can be applied while preserving soundness and completeness of the algorithm Add_BoundedPhasedRecovery in this paper. Furthermore, notice that the proof of Theorem 9.2.1 in its current form holds with this assumption. ∎

We now describe the algorithm $\mathsf{Add\_BoundedPhasedRecovery}$ in detail:

- (*Step 1*) First, we use the above technique to transform the input program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$ into a region graph $R(\mathcal{P}) = \langle \Pi^r_{\mathcal{P}}, Inv^r_{\mathcal{P}} \rangle$. To this end, we invoke the procedure $\mathsf{ConstructRegionGraph}$ as a black box (Line 1). We let this procedure convert state predicates and sets of transitions in $\mathcal{P}$ (e.g., $Inv_{\mathcal{P}}$ and $T_{\mathcal{P}}$) to their corresponding region predicates and sets of edges in $R(\mathcal{P})$ (e.g., $Inv^r_{\mathcal{P}}$ and $T^r_{\mathcal{P}}$).

- (*Step 2*) In order to ensure that the synthesized program does not violate $SPEC_{\overline{bt}}$, we identify the set ms of regions from where a computation may reach a transition in $SPEC_{bt}$ by taking fault transitions alone (Line 2). Next (Line 3), we compute the set mt of edges, which contains (1) edges that directly violate safety (i.e., $SPEC^r_{bt}$), and (2) edges whose target region is in ms (i.e., edges that lead a computation to a state from where safety may be violated by faults alone). Since the program does not have control over occurrence of faults, we remove the set ms from the region predicate S^r_1, which is our initial estimate of the fault-span (Line 4). Likewise, in Step 3, we will remove mt from the set of program edges $T^r_{\mathcal{P}}$ when recomputing program transitions.

- (*Step 3*) In this step, we add recovery paths to $R(\mathcal{P})$ so that $R(\mathcal{P})$ satisfies $\neg Inv_{\mathcal{P}'} \mapsto_{\leq \theta} Q$ and $Q \mapsto_{\leq \delta} Inv_{\mathcal{P}'}$. To this end, we first recompute the set $T_{\mathcal{P}_1}$ of program edges (Line 7) by including (1) existing edges that start and end in Inv^r_1, and (2) new *recovery edges* that originate from regions in $S^r_1 - Q^r$ (respectively, $Q^r - Inv^r_1$) and terminate at regions in S^r_1 (respectively, Q) such that the time-monotonicity condition is met. We exclude the set mt from $T^r_{\mathcal{P}_1}$ to ensure that these recovery edges do not violate $SPEC_{\overline{bt}}$. Notice that the algorithm allows arbitrary clock resets during recovery. If such clock resets are not desirable, one can rule them out by including them as bad transitions in $SPEC_{bt}$.

  After adding recovery edges, we invoke the procedure $\mathsf{Add\_rtUNITY}$ (Line 8) with parameters $S^r_1 - Q^r$, Q^r, and θ to ensure that $R(\mathcal{P})$ indeed satisfies the bounded response property $\neg Inv_{\mathcal{P}} \mapsto_{\leq \theta} Q$. The details of how the procedure $\mathsf{Add\_rtUNITY}$ (first proposed in Chapter 6) functions are not provided in this paper, with the exception of the following properties: (1) it adds a clock variable, say t_1, which gets reset when $S_1 - Q$ becomes true, to the set X of clock variables of $\mathcal{P}$, (2) for each state σ in $S_1 - Q$, it includes the set of transitions that participate in forming the computation that starts from σ and reaches a state in Q with smallest possible time delay, if the delay is less than θ, and (3) the regions made unreachable by this procedure (returned as the set ns) cannot be present in any solution that satisfies $\neg Inv_1 \mapsto_{\leq\theta} Q$. The procedure may optionally include additional computations, provided they preserve the corresponding bounded response property. Thus, since there does not exist a computation prefix that maintains the corresponding bounded response property from the regions in ns, in Line 9, the algorithm removes ns from S_1^r. Likewise, in Line 10, the algorithm adds a clock variable, say t_2, which gets reset when $Q - Inv_1$ becomes true and ensures that $R(\mathcal{P})$ satisfies $Q \mapsto_{\leq\delta} Inv_1$.

- (*Step 4*) Since we remove the set ns of regions from Inv_1^r, we need to ensure that S_1 is closed in F. Thus, we remove regions from where a sequence of fault edges can reach a region in ns (Lines 12-14). Next, due to the possibility of removal of some regions and edges in the previous steps, the algorithm ensures that the region graph $\langle \Pi^r_{\mathcal{P}_1}, Inv^r_{\mathcal{P}_1} \rangle$ does not have deadlock regions in the region invariant Inv_1^r (Lines 15-17). If the removal of deadlock regions and regions from where the closure of fault-span is violated results in empty invariant or fault-span, the algorithm declares failure (Lines 18-20).

- (*Step 5*) Finally, upon reaching a fixpoint, we transform the resulting region graph $\langle \Pi^r_{\mathcal{P}_1}, Inv^r_{\mathcal{P}_1} \rangle$ back into a real-time program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, Inv_{\mathcal{P}'} \rangle$ by invoking the procedure ConstructRealTimeProgram. In fact, the program $\mathcal{P}'$ is returned as the final synthesized fault-tolerant program. Note that since a region graph is a time-abstract *bisimulation* [AD94], we will not lose any behaviors in the reverse transformation.

We now show that the algorithm Add_BoundedPhasedRecovery is *sound* in the sense that any program that it synthesizes is correct-by-construction. We also show that the algorithm is *complete* in the sense that if it fails to synthesize a solution then no other correct solution exists.

Theorem 9.3.2 *The Algorithm Add_BoundedPhasedRecovery is sound.*

Proof. We show that the algorithm satisfies the constraints of Problem Statement 7.3.2. Let the algorithm add two clock variables t_1 and t_2 when invoking Add_rtUNITY (cf. Lines 8 and 10). We proceed as follows:

1. (*Constraints* $C1..C3$) By construction, correctness of these constraints trivially follows.

2. (*Constraint* $C4$) We distinguish two subgoals based on the behavior of $\mathcal{P}'$ in the absence and presence of faults:

   - We need to show that in the absence of faults, $\mathcal{P}' \models_{Inv_{\mathcal{P}'}} SPEC$. To this end, consider a computation $\overline{\sigma}$ of $T_{\mathcal{P}'}$ that starts in $Inv_{\mathcal{P}'}$. Since the values of t_1 and t_2 are of no concern inside $Inv_{\mathcal{P}'}$, from $C1$, $\overline{\sigma}$ starts from a state in $Inv_{\mathcal{P}}$, and from $C2$, $\overline{\sigma}$ is a computation of $T_{\mathcal{P}}$. Moreover, since we remove deadlock states from $Inv_{\mathcal{P}'}$ (cf. Lines 15-17), if $\overline{\sigma}$ is infinite in $\mathcal{P}$ then it is infinite in $\mathcal{P}'$ as well. It follows that $\overline{\sigma} \in SPEC$. Hence, every computation of $T_{\mathcal{P}'}$ that starts from a state in $Inv_{\mathcal{P}'}$ is in $SPEC$. Also, by construction, $Inv_{\mathcal{P}'}$ is closed in $T_{\mathcal{P}'}$. Furthermore, for all open regions, say r_0, in S'^r there exists an outgoing edge, say (r_0, r_1), for some $r_1 \in Inv^r_{\mathcal{P}'}$ where $r_0 \neq r_1$. Since the intolerant program exhibits no time-convergent behavior, such an edge can only terminate at a different clock region, which in turn advances time by an integer. This implies that in the absence of faults, our algorithm does not introduce time-convergent computations (*Zeno* behaviors) to $\mathcal{P}'$ and, hence, $\mathcal{P}' \models_{Inv_{\mathcal{P}'}} SPEC$.

   - Notice that by construction, $S_{\mathcal{P}'}$ is closed in $T_{\mathcal{P}'}[]F$ (cf. Lines 12-14). Now, we need to show that every computation of $T_{\mathcal{P}'}[]F$ that starts from a state in $S_{\mathcal{P}'}$ reaches a state in Q and subsequently a state in $Inv_{\mathcal{P}'}$ within θ and δ time units, respectively. Consider a computation $\overline{\sigma} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ of $T_{\mathcal{P}'}[]F$ that starts from a state in $S_{\mathcal{P}'}$. If $\sigma_0 \in Inv_{\mathcal{P}'}$ a single fault transition may take $\overline{\sigma}$ to $S_{\mathcal{P}'} - Inv_{\mathcal{P}'}$ and by Assumption 1, none of the subsequent transitions in $\overline{\sigma}$ is in F. Thus, $\overline{\sigma}_1 = (\sigma_1, \tau_1) \rightarrow (\sigma_2, \tau_2) \rightarrow \cdots$ is a computation of $T_{\mathcal{P}'}$ where $\sigma_1 \in S_{\mathcal{P}'} - Inv_{\mathcal{P}'}$. In the algorithm, when the repeat-until loop terminates, by construction, (1) $\overline{\sigma}_1$ reaches a state, say σ_k, in Q where $\tau_k - \tau_1 \leq \theta$ (cf. Lines 7-8), and (2) from σ_k, $\overline{\sigma}$ reaches a state in $Inv_{\mathcal{P}}$' within δ (cf. Line 10). This also implies that in the presence of faults as well, our algorithm does not introduce time-convergent computations to $\mathcal{P}'$, as $\overline{\sigma}$ eventually reaches a state in $Inv_{\mathcal{P}'}$. ∎

Theorem 9.3.3 *The Algorithm Add_BoundedPhasedRecovery is complete.*

Proof. The proof of completeness is based on the observation that if any state is removed then it must be removed, i.e., there is no fault-tolerant program that meets the constraints of Problem Statement 7.3.2 and includes this state. For example, in the computation of ms, if (σ_0, σ_1) is a fault transition and violates safety then state σ_0 must be removed (i.e., not reached). Likewise, ms includes states from where execution of faults alone violates safety. Hence, they must be removed. In Line 7, we compute the input program that includes all possible transitions that may be used in the final program. Due to constraint $C3$ of the Problem Statement 7.3.2, any transition that begins in the invariant must be a transition of the fault-intolerant program. Due to closure of Q in the sufficient condition, any transition from Q (precisely $Q - Inv_{\mathcal{P}}$ since states in $Inv_{\mathcal{P}}$ are already handled) must end in Q. And, due to closure of fault-span, any transition that begins in $S_{\mathcal{P}}$ (precisely $S_{\mathcal{P}} - Q$) must end in $S_{\mathcal{P}}$. Thus, the transitions computed in Line 7 are maximal. Furthermore, using the property of the Add_rtUNITY, if any state is removed in spite of considering all possible transitions that could be potentially used, then that state must be removed (i.e., states in ns).

Our algorithm declares failure when either the invariant or fault-span of the synthesized program is equal to the empty set. In other words, our algorithm fails to find a solution when all states of the intolerant program are illegitimate with respect to Problem Statement 7.3.2. Therefore, the algorithm Add_BoundedPhasedRecovery is complete. ∎

9.3.1 Example (cont'd)

We now demonstrate how the algorithm Add_BoundedPhasedRecovery synthesizes a fault-tolerant version of $\mathcal{TC}$, which provides bounded-time recovery. In the recovery specification of $\mathcal{TC}$ (cf. in Section 9.1), the invariant predicate $Inv_{\mathcal{TC}}$ and intermediate recovery predicate $Q_{\mathcal{TC}}$ were disjoint and in Section 9.2, we showed such specifications make the synthesis problem NP-complete. Now, let the intermediate recovery predicate be:

$$Q_{new} = Inv_{\mathcal{TC}} \ \cup \ Q_{\mathcal{TC}}.$$

In other words, after the occurrence of faults, the recovery specification requires that either both signals turn red within 3 time units and then return to the normal behavior within 7 time units, or, the system reaches a state in $Inv_{\mathcal{TC}}$ within 3 time units. Since, $Inv_{\mathcal{TC}} \subseteq Q_{new}$, we apply the Algorithm Add_BoundedPhasedRecovery to transform $\mathcal{TC}$ into a fault-tolerant program $\mathcal{TC}'$. We note that due to many symmetries in $\mathcal{TC}$ and the complex structure of the algorithm, we only present a highlight of the process of synthesizing $\mathcal{TC}'$.

First, observe that in Step 2 of the algorithm, $ms = \{\}$ and $mt = SPEC_{bt_{\mathcal{TC}}}$. In Step 3, consider a subset of $S_1 - Q_{new}$ where $(sig_0 = sig_1 = R) \wedge (z_0, z_1 \leq 1)$. This predicate is reachable by a single occurrence of (for instance) F_0 from an invariant state where $(sig_0 = sig_1 = R) \wedge (z_0 > 1) \wedge (z_1 \leq 1)$. After adding legitimate recovery transitions (Line 7), the invocation of Add_rtUNITY (Line 8) results in addition of the following recovery action:

$$\mathcal{TC}5_i \;::\; (sig_0 = sig_1 = R) \;\wedge\; (z_0, z_1 \leq 2) \;\wedge\; (t_1 \leq 2) \quad \longrightarrow \quad \mathbf{wait};$$

for all $i \in \{0, 1\}$. This action enforces the program to take delay transitions so that the program reaches a state in Q where $(sig_0 = sig_1 = R) \wedge (z_0, z_1 > 1)$.

Comment. One may notice that although it is perfectly legitimate to wait up to 3 time units inside $T_1 - Q_{new}$, as $\theta = 3$, the action $\mathcal{TC}5$ lets the program wait only for 2 time units. This is because Add_rtUNITY first includes computations with the smallest possible time delay and *optionally* includes additional computations to increase the level non-determinism. In this context, other such additional computations may be constructed by the following actions for sig_0:

$$\begin{array}{lllll}
\mathcal{TC}6_0 :: & (sig_0 = sig_1 = R) \;\wedge\; (z_0, z_1 \leq 1) \;\wedge\; (t_1 \leq 1) & \xrightarrow{\{x_0\}} & (sig_0 \;:=\; G); \\
\mathcal{TC}7_0 :: & (sig_0 = G) \wedge (sig_1 = R) \wedge & & \\
 & (x_0 = 1) \;\wedge\; (z_0, z_1 \leq 1) \;\wedge\; (t_1 \leq 2) & \xrightarrow{\{y_0\}} & (sig_0 \;:=\; Y); \\
\mathcal{TC}8_0 :: & (sig_0 = G) \wedge (sig_1 = R) \wedge & & \\
 & (x_0 \leq 1) \;\wedge\; (z_0, z_1 \leq 1) \;\wedge\; (t_1 \leq 2) & \longrightarrow & \mathbf{wait};
\end{array}$$

Notice that while executing recovery action $\mathcal{TC}6_0$ results in reaching another state in $S_1 - Q_{new}$, execution of actions $\mathcal{TC}7_0$ and $\mathcal{TC}8_0$ result in reaching a state in invariant $Inv_{\mathcal{TC}}$, which is clearly in Q_{new} as well. ∎

Now, consider the case where $\mathcal{TC}$ is in a state where $(sig_0 = G) \wedge (sig_1 = R) \;\wedge\; (x_0 = 1) \;\wedge\; (z_0, z_1 \leq 1)$. In this case, one may argue that $\mathcal{TC}$ has the option of executing action $\mathcal{TC}3_1$ and reaching a state where $sig_0 = sig_1 = G$, which is clearly a violation of safety specification $SPEC_{\overline{bt}_{\mathcal{TC}}}$. However, since we remove the set mt from $T_{\mathcal{P}_1}$ (Line 7), action $\mathcal{TC}3_i$ would be revised as follows:

$$\mathcal{TC}3_i :: \quad (sig_i = R) \;\wedge\; (z_j \leq 1) \;\wedge\; (sig_j \neq G) \quad \xrightarrow{\{x_i\}} \quad (sig_i := G);$$

for all $i \in \{0,1\}$ where $j = (i+1) \mod 2$. In other words, the algorithm strengthens the guard of $\mathcal{TC}1_i$ such that in the presence of faults, a signal does not turn green while the other one is also green.

In Step 4, consider the state predicate $Q_{new} - Inv_{1_{\mathcal{TC}}} = (sig_0 = sig_1 = R) \wedge (z_0, z_1 > 1)$. Similar to Step 3, the algorithm adds recovery paths with the smallest possible time delay, which is the following action for either $i = 0$ or $i = 1$:

$$\mathcal{TC}9_i{::} \quad (sig_i = sig_j = R) \ \wedge \ (z_i, z_j > 1) \quad \xrightarrow{\{z_i\}} \quad \textbf{skip};$$

It is straightforward to verify that by execution of $\mathcal{TC}9_i$, the program reaches the invariant $Inv_{\mathcal{TC}}$ from where the program behaves correctly. Similar to Step 3, the procedure Add_rtUNITY may include the following additional actions:

$$\begin{array}{llcl}
\mathcal{TC}10_i{::} & (sig_i = sig_j = R) \ \wedge \ (z_i, z_j > 1) \ \wedge \ (t_2 \leq 7) & \xrightarrow{\{x_i\}} & (sig_i := G); \\
\mathcal{TC}11_i{::} & (sig_i = sig_j = R) \ \wedge \ (z_i, z_j > 1) \ \wedge \ (t_2 \leq 7) & \xrightarrow{\{y_i\}} & (sig_i := Y); \\
\mathcal{TC}12_i{::} & (sig_i = sig_j = R) \ \wedge \ (z_i, z_j > 1) \ \wedge \ (t_2 \leq 7) & \longrightarrow & \textbf{wait};
\end{array}$$

Comment. One may notice that action $\mathcal{TC}11_i$ adds a strange behavior to $\mathcal{TC}$ by allowing a signal to change phase from red to yellow. Our algorithm allows addition of such recovery action, since it does not violate the safety specification $SPEC_{\overline{bt}_{\mathcal{TC}}}$. One may enforce the algorithm not to add such actions by simply adding the transitions in the set $\{(\sigma_0, \sigma_1) \mid \exists i \in \{0,1\} \ : \ (sig_i(\sigma_0) = R) \ \wedge \ (sig_i(\sigma_1) = Y)\}$ to $SPEC_{bt_{\mathcal{TC}}}$. In fact, we expect that our synthesis techniques have the potential to identify missing properties in cases where the specification is incomplete. ∎

In the context of $\mathcal{TC}$, in Step 5, the algorithm removes states from neither the fault-span nor the invariant, as $ns = \{\}$, and, hence, the algorithm finds the final solution in one iteration of the repeat-until loop.

Chapter 10

Disassembling Real-Time Fault-Tolerant Programs

Dependability and time-predictability are two vital properties of most embedded (especially, safety/mission-critical) systems. Consequently, providing *fault-tolerance* and meeting *timing constraints* are two inevitable aspects of dependable real-time embedded systems. Thus, it is highly desirable to have access to methodologies to formally reason about and, hence, gain assurance of these aspects during the design and analysis of embedded systems. Although in Part III of this dissertation, our focus is mainly on automated synthesis of such systems, their analysis is also of great interest. In the context of analysis, verification of fault-tolerant real-time embedded systems may be accomplished by illustrating the existence of constituents that (1) guarantee fault-tolerance, (2) ensure timing constraints, and (3) perform basic functionalities. One can also pose this problem as whether the structure of the programs that we synthesize is sensible. With this motivation, we focus on the following question:

> *Can a real-time fault-tolerant program be decomposed into components that can assist in its verification?*

In this chapter, we answer this question affirmatively by broadening *the theory of fault-tolerance components* [AK98b] to the context of real-time programs. The theory in [AK98b] essentially separates fault-tolerance and functionality concerns of untimed systems. More specifically, the theory identifies two types of fault-tolerance components, namely *detectors* and *correctors*. These components are based on the principle of detecting a *state predicate* to ensure that program actions would be safe and correcting a *state predicate* to ensure that the program eventually reaches a legitimate state. We emphasize that since these components do not rely on *detecting faults* or *correcting faults*, they can be applied in cases where faults are not detectable (e.g., Byzantine faults).

In the context of real-time programs, we focus on decomposition of *hard-masking* real-time fault-tolerant programs (cf. Definition 7.2.2), where (1) (timing independent) safety, (2) timing constraints, and (3) liveness properties (including recovery to legitimate states) are met even in the presence of faults. We identify three types of components, namely, *detectors*, *weak δ-correctors*, and *strong δ-correctors*. We show that these three components are in turn responsible for meeting the three properties of hard-masking fault-tolerant programs. Our proofs are constructive in the sense that they assist in identifying and subsequently decomposing a given hard-masking program into its fault-intolerant version, detectors, and δ-correctors.

Intuitively, detectors and δ-correctors work as follows. Each of these components is specified using two predicates: a *detection* (respectively, *correction*) predicate and a *witness* predicate. The goal of the detector component is to detect whether the given detection predicate is true and subsequently satisfies the witness predicate. It is required that whenever the witness predicate is true, the detection predicate must be true as well. Thus, the fault-tolerant program can use the witness predicate of the detector to provide the desired fault-tolerance requirements. In case of a δ-corrector, the component restores the program to a state where the correction predicate is true within a bounded amount of time. The use of the witness predicate by the program is optional, as the program may not need to know when the program state is restored.

Since we focus on demonstrating the *existence* of these components in a given hard-masking program, our notion of decomposition differs from that in [AK98b]. In particular, we precisely define what it means for a fault-tolerant program to *reuse* a fault-intolerant program. Furthermore, we formally define what it means for a fault-tolerant program to *contain* detectors and/or δ-correctors. We note that for assistance in analysis, it is necessary to ensure that the specification of the added components can be derived from the fault-intolerant version. Thus, in case of detectors, it is necessary to specify how the results of these components (intuitively, the conclusion that the predicate being detected is true) are *syntactically* used in the fault-tolerant real-time program. This part is not important in the context of a design methodology and, hence, is not formalized in [AK98b].

Although detectors and correctors have been found to be useful in the *design* of fault-tolerant programs [1], their significance in analysis has not been evaluated except in empirical case studies. In these studies [KRS99, GJ04], decomposition of a fault-tolerant program into its components has been found valuable in formal verification of the program. Thus, we expect that an affirmative answer to existence of the components would significantly assist in analysis of real-time embedded fault-tolerant programs.

The rest of this chapter is organized as follows. In Section 10.1, we present the basic concepts and assumptions related to this chapter. Then, in Section 10.2 (respectively, 10.3), we present the notion of detectors (respectively, δ-correctors) components, the concept of their containment in real-time programs, and their theory of decomposition.

10.1 Basic Concepts and Assumptions

In this section, we present some modifications to definitions presented in previous chapters. We start with computation. Notice that in Definition 2.1.7, we did not specify an initial value for the global time. Now, let Σ be any set of computations. We require that Σ must be closed with respect to *time offsets*. That is, $\forall \overline{\sigma} \in \Sigma \;:\; \forall t \in \mathbb{R} \;:\; (\overline{\sigma} + t) \in \Sigma$, where $\overline{\sigma} + t$ denotes the computation $(\sigma_0, \tau_0 + t) \rightarrow (\sigma_1, \tau_1 + t) \rightarrow \cdots$, st. $\tau_0 + t \geq 0$.

Notation. Let $\overline{\sigma}_i$ denote the pair (σ_i, τ_i) in computation $\overline{\sigma}$. Also, let $\overline{\alpha}$ be a finite computation of length n and $\overline{\beta}$ be a finite or infinite computation. The *concatenation* of $\overline{\alpha}$ and $\overline{\beta}$ (denoted $\overline{\alpha}\overline{\beta}$) is a computation, iff states $\overline{\alpha}_{n-1}$ and $\overline{\beta}_0$ meet the constraints of Definition 2.1.7. Otherwise, the result of concatenation is null. If Γ and Ψ are two sets containing finite and finite/infinte computations respectively, then $\Gamma\Psi = \{\overline{\alpha}\overline{\beta} \quad — \quad (\overline{\alpha} \in \Gamma) \;\wedge\; (\overline{\beta} \in \Psi)\;\}$.

Definition 10.1.1 (suffix and fusion closure) *Suffix closure* of a set of computations means that if a computation $\overline{\sigma}$ is in that set then so are all the suffixes of $\overline{\sigma}$. *Fusion closure* of a set of computations means that if computations $\overline{\alpha}(\sigma, \tau)\overline{\gamma}$ and $\overline{\beta}(\sigma, \tau)\overline{\psi}$ are in that set then so are the computations $\overline{\alpha}(\sigma, \tau)\overline{\psi}$ and $\overline{\beta}(\sigma, \tau)\overline{\gamma}$, where $\overline{\alpha}$ and $\overline{\beta}$ are computation prefixes, $\overline{\gamma}$

[1]The components have been shown to suffice in the design of a large class of fault-tolerant programs [AK98b] including programs designed using replication, state machine approach, and, checkpointing and recovery.

and $\overline{\psi}$ are computation suffixes, and σ is a state at global time τ. ∎

Definition 10.1.2 (**S-computations**) Let S be a state predicate and $\mathcal{P} = \langle \Pi_\mathcal{P}, Inv_\mathcal{P} \rangle$ be a program. The S-computations of $\mathcal{P}$, denoted as $\mathcal{P} \triangleright S$, is the set of all computations in $\Pi_\mathcal{P}$ that start in a state where S is true. ∎

In this chapter, we assume that all sets of computations are suffix and fusion closed. Also, in this chapter, we relax the definition of computations (see Definition 2.1.7) by allowing them to exhibit time-divergent behavior as well. One can observe that removing the time-convergence restriction allows real-time programs and specifications to exhibit *Zeno* behavior where a program is forced into a corner to take infinite number of steps within a finite amount of time. The reason for this modification in this chapter is that when we develop the theory of fault-tolerance components in Sections 10.2 and 10.3, we allow components to exhibit Zeno behavior. Such modeling gives freedom in dealing with situations where a component reaches a state from where another component must make progress. However, as we will illustrate, it is important that the collection of components working together in a program does not exhibit Zeno behavior. We emphasize that all results in this paper are independent of Zeno or nonZeno constraints on components.

We now redefine what it means for a program to *refine* a specification and what it means for a program $\mathcal{P}'$ (typically, a fault-tolerant program) to refine a program $\mathcal{P}$ (typically, a fault-intolerant program). Essentially, we would like to say that '$\mathcal{P}'$ refines $\mathcal{P}$' iff computations of $\mathcal{P}'$ are a subset of that in $\mathcal{P}$. However, if $\mathcal{P}'$ is obtained by adding fault-tolerance to $\mathcal{P}$ then $\mathcal{P}'$ may contain additional variables that are not in $\mathcal{P}$. Hence, it will be necessary to project the computations of $\mathcal{P}'$ on (the variables of) $\mathcal{P}$ and then check if the projected computation is a computation of $\mathcal{P}$. Recall that $V_\mathcal{P}$ and $X_\mathcal{P}$ denote the set of discrete and clock variables of program $\mathcal{P}$, respectively.

Definition 10.1.3 (**state projection**) Let $\mathcal{P}$ and $\mathcal{P}'$ be real-time programs st. $V_{\mathcal{P}'} = V_\mathcal{P} \cup \Delta_v$ and $X_{\mathcal{P}'} = X_\mathcal{P} \cup \Delta_x$ for some Δ_v and Δ_x. The *projection* of a state of $\mathcal{P}'$ on $\mathcal{P}$ is a state obtained by considering $V_\mathcal{P} \cup X_\mathcal{P}$ only, i.e., by abstracting away the variables in $\Delta_v \cup \Delta_x$. ∎

The same concept applies to programs and specifications. Extending this definition for computations, we say that the projection of a computation of $\mathcal{P}'$ on $\mathcal{P}$ (respectively, $SPEC$)

is a computation obtained by projecting each state in that computation on $\mathcal{P}$ (respectively, *SPEC*).

Definition 10.1.4 (refines) Let $\mathcal{P}$ and $\mathcal{P}'$ be real-time programs, S be a state predicate and *SPEC* be a specification. We say that $\mathcal{P}'$ *refines* $\mathcal{P}$ (respectively, *SPEC*) *from* S iff the following two conditions hold: (1) S is closed in $\mathcal{P}'$, and (2) for every computation of $\mathcal{P}'$ that starts in a state where S is true, the projection of that computation on $\mathcal{P}$ (respectively, *SPEC*) is a computation of $\mathcal{P}$ (respectively, *SPEC*. ∎

Specifying timing constraints. In order to express time-related behaviors of real-time programs (e.g., deadlines and recovery time), in this chapter, we focus on a special type of bounded response properties known as *stable bounded response property* (equivalent to UNITY bounded-time ensures properties). A stable bounded response property, denoted $P \mapsto_{\leq \delta} Q$, where P and Q are two state predicates and $\delta \in \mathbb{Z}_{\geq 0}$, is the set of all computations $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ in which, for all $i \geq 0$, if $\sigma_i \models P$ then there exists j, $j \geq i$, st. (1) $\sigma_j \models Q$, (2) $\tau_j - \tau_i \leq \delta$, and (3) for all k, $i \leq k < j$, $\sigma_k \models P$, i.e., it is always the case that a state in P is followed by a state in Q within δ time units and P remains true until Q becomes true. We call P the *event predicate*, Q the *response* (or *recovery*) *predicate*, and δ the *response* (or *recovery*) *time*.

Assumption 10.1.5 We assume that the set of clock variables of any stable bounded response property $P \mapsto_{\leq \delta} Q$ contains a special clock variable, which gets reset whenever P becomes true. This assumption is necessary to ensure that stable bounded response properties are fusion closed. ∎

10.2 Detectors and Their Role in Hard-Masking Programs

This section is organized as follows. We formally introduce the notion of detector components in Subsection 10.2.1. In Subsection 10.2.2, we precisely define what we mean by containment of a detector in a real-time program. Then, we present detector components of the hard-masking version of our traffic controller in Subsection 10.2.3. Finally, in Subsection 10.2.4, we develop the theory of detectors by proving the necessity of existence of hard-masking detectors in hard-masking fault-tolerant programs.

10.2.1 Detectors

Intuitively, a detector is a program component that ensures satisfaction of timing independent safety (i.e., $SPEC_{\overline{bt}}$ in Definition 2.2.4).

Definition 10.2.1 (detects) Let W and D be state predicates. Let 'W *detects* D' be the specification, that is the set of all infinite computations $\overline{\sigma} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$, satisfying the following three conditions:

- (*Safeness*) For all $i \in \mathbb{Z}_{\geq 0}$, if $\sigma_i \models W$ then $\sigma_i \models D$. (In other words, $\sigma_i \models (W \Rightarrow D)$.)
- (*Progress*) For all $i \in \mathbb{Z}_{\geq 0}$, if $\sigma_i \models D$ then there exists $k, k \geq i$, st. $\sigma_k \models W$ or $\sigma_k \not\models D$.
- (*Stability*) There exists $i \in \mathbb{Z}_{\geq 0}$, st. for all j, $j \geq i$, if $\sigma_j \models W$ then $\sigma_{j+1} \models W$ or $\sigma_{j+1} \not\models D$. ∎

Definition 10.2.2 (detectors) Let $\mathcal{D}$ be a program and D, W, and U be state predicates of $\mathcal{D}$. We say that W *detects* D *in* $\mathcal{D}$ *from* U (i.e., $\mathcal{D}$ is a *detector*) iff $\mathcal{D}$ refines 'W detects D' from U. ∎

A detector $\mathcal{D}$ is used to check whether its "detection predicate", D, is true. Since $\mathcal{D}$ satisfies *Progress* from U, in any computation of $\mathcal{D}$, if $U \wedge D$ is true continuously, $\mathcal{D}$ eventually detects this fact and makes W true. Since $\mathcal{D}$ satisfies *Safeness* from U, it follows that $\mathcal{D}$ never lets W witness D incorrectly. Moreover, since $\mathcal{D}$ satisfies *Stability* from U, it follows that once W becomes true, it continues to be true unless D is falsified. In the context of fault-tolerance, D is typically a predicate of the fault-intolerant program from where safety should be always satisfied and W is a predicate of the fault-tolerant program that witnesses the detection of D.

In order to analyze the behavior of a detector in the presence of faults, we consider the notion of hard-masking tolerant detectors. More specifically, a detector $\mathcal{D}$ is a *hard-masking tolerant detector*.

10.2.2 Containment of Detectors in Real-Time Programs

In order to show the existence of detectors in hard-masking fault-tolerant programs, we would like to show that the program contains a detector for a detection predicate associated with the fault-intolerant program. However, we need to identify syntactic characteristics of a program before detection predicates can be identified. In particular, since a detector is used to ensure that the execution of an action is safe, its witness predicate must be *used* by the fault-tolerant program. Intuitively, the syntactic constraints identified in this section require the witness predicate to be a guard of the corresponding action in the fault-tolerant program.

In order to accomplish our goal, first, we show that violation of timing independent safety (i.e., $SPEC_{\overline{bt}}$ in Definition 7.1.6) can be merely determined by considering transitions in $SPEC_{bt}$.

Lemma 10.2.3 *Let $SPEC$ be a specification, $\overline{\alpha}$ be a computation prefix, σ and σ' be two states, and $\tau, \tau' \in \mathbb{R}_{\geq 0}$, where $\tau = \tau'$.*
If

- *$\overline{\alpha}(\sigma, \tau)$ maintains $SPEC_{\overline{bt}}$*

then

- *$\overline{\alpha}(\sigma, \tau)(\sigma', \tau')$ maintains $SPEC_{\overline{bt}}$ iff $(\sigma, \tau)(\sigma', \tau')$ maintains $SPEC_{\overline{bt}}$.*

Proof. We distinguish two cases:

- ($\Rightarrow$) If $\overline{\alpha}(\sigma, \tau)(\sigma', \tau')$ maintains $SPEC_{\overline{bt}}$ then by definition of maintains (cf. Definition 7.1.4), there exists a computation suffix $\overline{\beta}$ st. $\overline{\alpha}(\sigma, \tau)(\sigma', \tau')\overline{\beta} \in SPEC_{\overline{bt}}$. Moreover, by suffix closure of $SPEC_{\overline{bt}}$, $(\sigma, \tau)(\sigma', \tau')\overline{\beta}$ must be in $SPEC_{\overline{bt}}$ as well. Hence, by definition of maintains, $(\sigma, \tau)(\sigma', \tau')$ maintains $SPEC_{\overline{bt}}$.

- ($\Leftarrow$) Consider the case where $\overline{\alpha}(\sigma, \tau)$ maintains $SPEC_{\overline{bt}}$ and $(\sigma, \tau)(\sigma', \tau')$ maintains $SPEC_{\overline{bt}}$. By definition of maintains there exists a suffix $\overline{\beta}$ st. $\overline{\alpha}(\sigma, \tau)\overline{\beta}$ is in $SPEC_{\overline{bt}}$ and there exists $\overline{\beta}'$ st. $(\sigma, \tau)(\sigma', \tau')\overline{\beta}'$ is in $SPEC_{\overline{bt}}$. Moreover, by fusion closure of $SPEC_{\overline{bt}}$, $\overline{\alpha}(\sigma, \tau)(\sigma', \tau')\overline{\beta}'$ is also in $SPEC_{\overline{bt}}$. Thus, by definition of maintains, $\overline{\alpha}(\sigma, \tau)(\sigma', \tau')$ maintains $SPEC_{\overline{bt}}$. ∎

In Lemma 10.2.4, we show that there exists a set of states from where execution of programs maintains $SPEC_{\overline{bt}}$. We call such a state predicate a *detection predicate* for $SPEC_{\overline{bt}}$.

Lemma 10.2.4 *Given a program $\mathcal{P}$, there exists a state predicate D (called* detection predicate*) st. all computations of $\mathcal{P}$ that start from D maintain $SPEC_{\overline{bt}}$.*

Proof. First, the set of computations of $\mathcal{P}$ is suffix and fusion closed, $\mathcal{P}$ can be represented in terms of a set of *transitions* (and consequently timed actions) that it can execute. Now, consider a computation prefix of $\Pi_{\mathcal{P}}$, say $\overline{\alpha}(\sigma, \tau)$, that maintains $SPEC_{\overline{bt}}$. Observe that from (σ, τ) execution of (a timed action of) $\mathcal{P}$ maintains $SPEC_{\overline{bt}}$ iff the extended prefix $\overline{\alpha}(\sigma, \tau)(\sigma', \tau')$, after execution of $\mathcal{P}$ maintains $SPEC_{\overline{bt}}$. Notice that since (σ', τ') is not reached by a delay action, $\tau = \tau'$. In other words, there exists a set of computation prefixes, say Γ, from which execution of $\mathcal{P}$ maintains $SPEC_{\overline{bt}}$.

From Lemma 10.2.3, it follows that the extended prefix $\overline{\alpha}(\sigma, \tau)(\sigma', \tau')$ maintains $SPEC_{\overline{bt}}$ iff $(\sigma, \tau)(\sigma', \tau')$ maintains $SPEC_{\overline{bt}}$. Thus, the execution of $\mathcal{P}$ maintains $SPEC_{\overline{bt}}$ iff it execute in a state in the set $\{(\sigma, \tau) \mid \exists \overline{\alpha} \ : \ \overline{\alpha}(\sigma, \tau) \in \Gamma)\}$. The predicate characterized by this set of states suffices as a witness for the lemma (i.e., predicate D). ∎

We now prove the uniqueness of the weakest detection predicate for a given program $\mathcal{P}$.

Lemma 10.2.5 *Given a program $\mathcal{P}$ and a specification $SPEC$, there exists a unique weakest detection predicate of $\mathcal{P}$ for $SPEC_{\overline{bt}}$.*

Proof. From Lemma 10.2.4, observe the existence of a detection predicate, and that a program may have multiple detection predicates. Formally, let wdp be a detection predicate of $\mathcal{P}$ for $SPEC_{\overline{bt}}$ and D be an arbitrary state predicate. It is easy to see that if $D \Rightarrow wdp$, then D is also a detection predicate of $\mathcal{P}$ for $SPEC_{\overline{bt}}$. Moreover, if wdp_1 and wdp_2 are two detection predicates of $\mathcal{P}$ for $SPEC_{\overline{bt}}$ then so is $wdp_1 \vee wdp_2$. Thus, there exists a unique weakest detection predicate for the given program. ∎

We are now ready to define what it means for a program to *contain* detectors. Given a timed guarded command, say $L \ :: \ g \xrightarrow{\lambda} st$, Lemma 10.2.5 shows that there exists a unique weakest detection predicate, say wdp, from where execution of L does not violate $SPEC_{\overline{bt}}$. Hence, to show the existence of detectors, we require the detection predicate of such a timed action to be $g \wedge wdp$. Furthermore, to show that the fault-tolerant program contains the desired detector, we show that it must be using the witness predicate of that detector to ensure that execution of the corresponding timed action is safe. Towards this end, we define the notion of *encapsulation*. Intuitively, if (typically, a fault-tolerant) program $\mathcal{P}'$ encapsulates fault-intolerant) program $\mathcal{P}$ then for each timed action of $\mathcal{P}$ of the form $g \xrightarrow{\lambda} st$, $\mathcal{P}'$ contains a timed action of the form $g \wedge g' \xrightarrow{\lambda \cup \lambda'} st||st'$. The semantic of $st||st'$ corresponds to the statement where st and st' are executed simultaneously, clock variables in $\lambda \cup \lambda'$ are reset, and the timed action is executed only when its guard, $g \wedge g'$, is true. In other words, $\mathcal{P}'$ has a timed action corresponding to each timed action of $\mathcal{P}$ (possibly) with a stronger guard, additional assignments in st', and additional clock variables in λ'. Notice that the assignments in st' and clock variables in λ' may be added in order to add fault-tolerance to $\mathcal{P}$ (cf. the notion of state projection in Definition 10.1.3). To show that $\mathcal{P}'$ is using a detector for a timed action of $\mathcal{P}$, we require the witness predicate of that detector to be $g \wedge g'$ which is the guard of the corresponding timed action in $\mathcal{P}'$.

Definition 10.2.6 (encapsulates) Let $\mathcal{P}$ and $\mathcal{P}'$ be two real-time programs and S be a state predicate. We say that $\mathcal{P}'$ *encapsulates* $\mathcal{P}$ *from* S iff each timed action in $\mathcal{P}'$ that is enabled in a state in S and that updates variables in $V_{\mathcal{P}}$ is of the form $g \wedge g' \xrightarrow{\lambda \cup \lambda'} st||st'$, where $g \xrightarrow{\lambda} st$ is a timed action of $\mathcal{P}$ and st' does not update variables in $V_{\mathcal{P}}$ and $\lambda' \cap X_{\mathcal{P}} = \{\}$. ∎

Based on the above discussion, given a timed guarded command of the form $g \xrightarrow{\lambda} st$ of $\mathcal{P}$, its (weakest) detection predicate *wdp* and the corresponding action $g \wedge g' \xrightarrow{\lambda \cup \lambda'} st||st'$ of $\mathcal{P}'$, we require the detection predicate of the desired detector to be $g \wedge wdp$ and the witness predicate of the desired detector to be $g \wedge g'$.

Finally, in order to formalize the notion of containment and existence of detectors, we need to define what it means to obtain a fault-tolerant program by *reusing* its fault-intolerant version.

Definition 10.2.7 (reuses) Let $\mathcal{P}$ and $\mathcal{P}'$ be two real-time programs. We say that $\mathcal{P}'$ *reuses* $\mathcal{P}$ *from* S iff the following two conditions are satisfied:

- $\mathcal{P}'$ refines $\mathcal{P}$ from S, and
- $\mathcal{P}'$ encapsulates $\mathcal{P}$ from S. ∎

10.2.3 Example (cont'd)

Let $\mathcal{TC}$ be in a state where $(sig_0 = sig_1 = R) \wedge (z_0 > 1) \wedge (z_1 \leq 1)$. In this case, if fault F_0 occurs, subsequent execution of $\mathcal{TC}3_0$ and $\mathcal{TC}3_1$ results reaching a state where

$sig_0 = sig_1 = G$, which is clearly a violation of $SPEC_{\overline{bt}_{\mathcal{TC}}}$. It is straightforward to see that the weakest detection predicate for $\mathcal{TC}_i$ is:

$$wdp_{\mathcal{TC}'_i} = \{\sigma \mid sig_j(\sigma) = R\},$$

where $i \in \{0, 1\}$ and $j = (i + 1) \mod 2$. Thus, in program $\mathcal{TC}'$, the guard of $\mathcal{TC}3_i$ is strengthened in order for $\mathcal{TC}'$ to refine $SPEC_{\overline{bt}_{\mathcal{TC}}}$ in the presence of faults. Intuitively, $\mathcal{TC}'$ is allowed to change phase from red to green only when the other signal is red. More precisely, $\mathcal{TC}'$ uses the detector $\mathcal{D}_{\mathcal{TC}'_i}$ which consists of timed guarded commands $\mathcal{TC}'1_i$, $\mathcal{TC}'2_i$, and $\mathcal{TC}'4_i$ with the following detection and witness predicates:

$$D_{\mathcal{TC}'_i} = guard(\mathcal{TC}3_i) \wedge wdp_{\mathcal{TC}'_i}$$
$$W_{\mathcal{TC}'_i} = guard(\mathcal{TC}'3_i).$$

It is easy to see that $W_{\mathcal{TC}'_i}$ detects $D_{\mathcal{TC}'_i}$ in $\mathcal{D}_{\mathcal{TC}'_i}$ from $Inv_{\mathcal{TC}}$ in both absence and presence of F_0 and F_1. Observe that in this example, the witness and detection predicates happen to be equal. However, as we will show in the proof of Claim 10.2.10, this is not always the case. Notice that $\mathcal{D}_{\mathcal{TC}'_i}$ exhibits Zeno behavior since when the witness predicate becomes true, there does not exist a timed guarded command whose guard is enabled except $\mathcal{TC}'4_i$. However, as explained in Section 10.1, it is important that the entire program does not show Zeno behavior. For instance, one can observe that, $\mathcal{TC}'3_i$ ensures time progress for $\mathcal{TC}'$.

10.2.4 The Necessity of Existence of Detectors in Hard-Masking Programs

Based on the formalization of the notion of containment, we are now ready to prove that hard-masking programs contain hard-masking tolerant detectors. Our strategy to accomplish our goal is as follows. First, based on Definitions 10.2.6 and 10.2.7, we show that if a program refines $SPEC_{\overline{bt}}$ *in the absence of faults* then it contains detectors. The intuition is that if program $\mathcal{P}'$ is designed by transforming $\mathcal{P}$ so as to refine $SPEC_{\overline{bt}}$, then the transformation must have added detectors for $\mathcal{P}$, and $\mathcal{P}'$ reuses $\mathcal{P}$. We formulate this in Claim 10.2.10. Then (*in the presence of faults*), using Claim 10.2.10, we show that if a hard-masking program $\mathcal{P}'$ is designed by reusing $\mathcal{P}$ to tolerate a set F of faults, $\mathcal{P}'$ contains a hard-masking tolerant detector for each action of $\mathcal{P}$. This is shown in Theorem 10.2.11.

In order to show that a program contains a detector component, we are required to show that the corresponding timed guarded commands satisfy the *Progress* condition of Definition 10.2.1. Thus, we assume that programs need to satisfy the following *fairness* condition.

Assumption 10.2.8 We assume that program computations are *fair* in the sense that in every computation, if the guard of an action is continuously true then that action is eventually chosen for execution. ∎

Assumption 10.2.9 Without loss of generality, for simplicity, we assume that transitions that correspond to different actions of the program are mutually disjoint, i.e., they do not contain overlapping transitions. The results in this paper are valid without this assumption since we can easily modify a given program to one that satisfies this assumption. ∎

We are now ready to formulate our claim on existence of detectors in programs that refine $SPEC_{\overline{bt}}$ in the absence of faults.

Claim 10.2.10 *Let $\mathcal{P}$ and $\mathcal{P}'$ be real-time programs, Inv be a nonempty state predicate, and $SPEC$ be a specification.*
If

- *$\mathcal{P}'$ reuses $\mathcal{P}$ from Inv, and*
- *$\mathcal{P}'$ refines $SPEC_{\overline{bt}}$ from Inv,*

then

- *$(\forall ac \mid ac$ is a timed action of $\mathcal{P}$: $\mathcal{P}'$ contains a detector of a detection predicate of ac for $SPEC)$.*

Proof. Let ac be a timed action of $\mathcal{P}$. We show that $\mathcal{P}'$ contains a detector of a detection predicate of ac. Let wdp be the weakest detection predicate for ac. Since $\mathcal{P}'$ reuses (and, hence, encapsulates) $\mathcal{P}$ from Inv, if ac is of the form $g \xrightarrow{\lambda} st$, $\mathcal{P}'$ contains a timed action, say ac', of the form $g \wedge g' \xrightarrow{\lambda \cup \lambda'} st || st'$. Now, we instantiate witness predicate W and detection predicate D as follows:

$W = g \ \wedge \ g'$, and
$D = g \ \wedge \ wdp$.

Following Lemma 10.2.4, since $D \Rightarrow wdp$, whenever D is true, execution of ac maintains $SPEC_{\overline{bt}}$. Hence, D is a detection predicate of ac. Thus, it only remains to show that $\mathcal{P}'$ refines 'W detects D' from Inv. To this end, we verify Safeness, Progress, and Stability constraints of our detector:

- (*Safeness*) By definition of W, $W \Rightarrow g$. Since $\mathcal{P}'$ refines $SPEC_{\overline{bt}}$ from Inv, whenever ac is executed in a state where Inv is true, its execution maintains $SPEC_{\overline{bt}}$. Since wdp is the weakest detection predicate of ac, $Inv \wedge W \Rightarrow wdp$. Thus, $W \Rightarrow D$ and, hence, Safeness is satisfied.

- (*Progress*) Consider any computation, say $\overline{\sigma}'$, of $\mathcal{P}'$ which starts in a state where Inv is true, and, D is true in each state in $\overline{\sigma}'$. By definition of D, g is true in each state in $\overline{\sigma}'$. Now, consider the computation, say $\overline{\sigma}$, obtained by projecting $\overline{\sigma}'$ on $\mathcal{P}$. Since $\mathcal{P}'$ refines $\mathcal{P}$ from Inv, $\overline{\sigma}$ is a computation of $\mathcal{P}$ and g must be continuously true. Hence by fairness (cf. Assumption 10.2.8), action ac must eventually be executed. Let σ denote the state where timed action ac executes in $\overline{\sigma}$, and let σ' denote the corresponding state in $\overline{\sigma}'$. Consider the action ac' executed by $\mathcal{P}'$ in state σ'. Since we assume that the transitions included in different timed actions are mutually exclusive, ac' is the only action that can be executed at σ'. In other words, no other action in $\mathcal{P}'$ has the same effect on variables of $\mathcal{P}$ from state σ. Thus, W is true in state σ' and, therefore, Progress is satisfied.

- (*Stability*) Let ac and ac' be as defined above. If Stability does not hold in $\mathcal{P}'$, it implies there exists a computation of $\mathcal{P}'$ where the action ac' is never executed. Now, consider this computation of $\mathcal{P}'$ and its projection on $\mathcal{P}$. In the projected computation, g is continuously true. However, action ac is not executed. And, this is a contradiction since $\mathcal{P}'$ reuses $\mathcal{P}$ from Inv. ∎

Now, we show that if a hard-masking F-tolerant program $\mathcal{P}'$ is designed by reusing $\mathcal{P}$ then $\mathcal{P}'$ contains a hard-masking tolerant detector for each action in $\mathcal{P}$.

Theorem 10.2.11 *Let $\mathcal{P}$ and $\mathcal{P}'$ be real-time programs, Inv be a nonempty state predicate, F be a set of faults, and SPEC be a specification.*
If

- *$\mathcal{P}$ refines $SPEC_{\overline{bt}}$ from Inv,*
- *$\mathcal{P}'$ reuses $\mathcal{P}$ from R, where $R \Rightarrow Inv$ for some nonempty state predicate R, and*
- *$\mathcal{P}'$ is hard-masking F-tolerant to SPEC from R*

then

- *$(\forall ac \mid ac$ is a timed action of $\mathcal{P}$: $\mathcal{P}'$ is a hard-masking F-tolerant detector of a detection predicate of ac for SPEC).*

Proof. Let ac be a timed action of $\mathcal{P}$ and let wdp be the weakest detection predicate for ac. Since $\mathcal{P}'$ reuses (and, hence, encapsulates) $\mathcal{P}$ from R, if ac is of the form $g \xrightarrow{\lambda} st$ then $\mathcal{P}'$ contains a timed action, say ac', of the form $g \wedge g' \xrightarrow{\lambda \cup \lambda'} st||st'$. Now, for each timed action ac of $\mathcal{P}$, we instantiate witness predicate W and detection predicate D as follows:

$W = g \ \wedge \ g'$, and
$D = g \ \wedge \ wdp$.

Following Claim 10.2.10, it is straightforward to show that in the absence of faults, $\mathcal{P}'$ refines 'W detects D' from R. For the case where faults may perturb computations of $\mathcal{P}'$, we show that $\mathcal{P}'$ contains a hard-masking tolerant detector to $SPEC$. To this end, we need to show that there exists an F-span S st. $\mathcal{P}'[]F$ refines the hard-masking tolerance specification of 'W detects D' from S. To this end, we let S be the F-span used to show that $\mathcal{P}'$ is hard-masking f-tolerant for $SPEC$ from R. Now, we only need to show that all computations of $\mathcal{P}'[]F$ satisfy Safeness, Progress, and Stability. The proof of this part of the theorem is identical to the proof of Safeness, Progress, and Stability in Claim 10.2.10. ∎

10.3 δ-Correctors and Their Role in Hard-Masking Programs

This section is organized as follows. We formally introduce the notion of weak and strong δ-corrector components in Subsection 10.3.1. In Subsection 10.3.2, we define what we mean by containment of a δ-corrector in a real-time program. Then, we present δ-corrector components of the hard-masking version of our traffic controller in Subsection 10.3.3. Finally, in Subsections 10.3.4 and 10.3.5, we develop the theory of strong and weak δ-correctors, respectively, by proving the necessity of existence of hard-masking weak and strong δ-correctors in hard-masking fault-tolerant programs.

10.3.1 Weak and Strong δ-Correctors

Intuitively, a δ-corrector is a program component that ensures *bounded-time recovery* to a *correction predicate*. In fault-tolerant computing, recovery is essential to guarantee that liveness properties (cf. Definition 2.2.5) and timing constraints (cf. $SPEC_{\overline{br}}$ in Definition 2.2.4) are met where the state of a program is perturbed by the occurrence of faults. Depending upon the closure of the correction predicate in δ-correctors, they are classified into *weak* and *strong*.

Definition 10.3.1 (weakly corrects) Let C and W be state predicates. Let 'W *weakly corrects* C *within* δ' be the specification, that is the set of all infinite computations $\overline{\sigma} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$, satisfying the following conditions:

- (*[Weak] Convergence*) There exists $i \in \mathbb{Z}_{\geq 0}$, st. $\sigma_i \models C$ and $(\tau_i - \tau_0) \leq \delta$.
- (*Safeness*) For all $i \in \mathbb{Z}_{\geq 0}$, if $\sigma_i \models W$ then $\sigma_i \models C$.
- (*Progress*) For all $i \in \mathbb{Z}_{\geq 0}$, if $\sigma_i \models C$ then there exists $k, k \geq i$, st. $\sigma_k \models W$ or $\sigma_k \not\models C$.
- (*Stability*) There exists $i \in \mathbb{Z}_{\geq 0}$, st. for all j, $j \geq i$, if $\sigma_j \models W$ then $\sigma_{j+1} \models W$ or $\sigma_{j+1} \not\models C$. ∎

Definition 10.3.2 (strongly corrects) Let C and W be state predicates. Let 'W *strongly corrects* C *within* δ' be the specification, that is the set of all infinite computations $\overline{\sigma}$, satisfying the following two conditions:

- W weakly corrects C within δ, and

- (*[Strong] Convergence*) In addition to Weak Convergence, C is closed in $\overline{\sigma}$. ∎

Definition 10.3.3 (**δ-correctors**) Let $\mathcal{C}$ be a program and C, W, and U be state predicates of $\mathcal{C}$. We say that W *weakly/strongly corrects* C *in* $\mathcal{C}$ *from* U (i.e., $\mathcal{C}$ is a weak/strong δ-*corrector*) iff $\mathcal{C}$ refines 'W weakly/strongly corrects C within δ' from U. ∎

Similar to the concept of tolerant detectors, in order to analyze the behavior of a δ-corrector $\mathcal{C}$ in the presence of faults, we consider the notion of hard-masking tolerant δ-correctors.

Notice that since $\mathcal{C}$ satisfies Weak (respectively, Strong) Convergence from U, it follows that $\mathcal{C}$ reaches a state where C becomes true within δ time units (and, respectively, C continues to be true thereafter). In addition to convergence, a δ-corrector never lets the predicate W witness the correction predicate C incorrectly, as $\mathcal{C}$ satisfies Safeness from U. Moreover, since $\mathcal{C}$ satisfies Progress from U, it follows that W eventually becomes true. And, finally, since $\mathcal{C}$ satisfies Stability from U, it follows that when W becomes true, W is never falsified.

10.3.2 Containment of δ-Correctors in Real-Time Programs

As mentioned earlier, in the context of real-time programs, δ-correctors ensure bounded-time recovery to their correction predicate. Intuitively, we will use weak δ-correctors where we need refinement of stable bounded response properties in the presence of faults. In such properties, when the event predicate becomes true, the program needs to reach a state where the response predicate holds within the respective recovery time. Nonetheless, the program does not need to remain in the response predicate.

Unlike weak δ-correctors, we will use strong δ-correctors where we need bounded-time recovery to a state predicate in which the program is required to stay in. The correction predicate of a δ-corrector $\mathcal{C}$ is typically an invariant predicate of the fault-intolerant program while the witness predicate witnesses the correction of the correction predicate. This is obviously due to the fact that real-time programs are closed in their invariant predicate. Existence of strong δ-correctors are of special interest, since recovery to the invariant predicate automatically ensures refinement of the liveness specification. In particular, in

Subsection 10.3.4, we show the necessity of existence of strong δ-correctors in hard-masking programs in order to refine the property $\neg Inv \mapsto_{\leq \theta} Inv$, where Inv is in invariant predicate.

In terms of the behavior of δ-correctors, observe that in Definition 10.3.1 (and, hence, Definition 10.3.2 as well), state σ_0 is the earliest state from where recovery must commence. Thus, τ_i is the time instance where correction is complete and $\tau_i - \tau_0$ is the duration of correction. In case of strong δ-correctors, σ_0 is also the earliest state reached outside the invariant due to occurrence of faults.

We note that although detectors and δ-correctors share three identical constraints, the semantics of their containment in real-time programs are completely different. Intuitively, a detector uses the witness predicate in order to detect whether program execution is safe. Hence, as we developed the theory of detectors in Section 10.2, we imposed constraints that require the witness predicate to be *used*. To the contrary, a program may not use the witness predicate of a δ-corrector, as a fault-tolerant program may not need to know *when* correction is complete.

10.3.3 Example (cont'd)

Continuing with our traffic controller example, we identify δ-correctors for each stable bounded-response property in $SPEC_{\overline{br}_{\mathcal{TC}}}$ introduced in Subsection 7.2.3. To this end, first, consider the property $\neg S_{\mathcal{TC}} \mapsto_{\leq 3} Q$. When $\mathcal{TC}[]\{F_0, F_1\}$ reaches a state in $\neg S_{\mathcal{TC}} \wedge \neg Q$ where at least one signal is red and the value of both z timers is less than or equal to 1, it needs to recover to Q within 3 time units. Let $\mathcal{C}^1_{\mathcal{TC}'}$ be the weak 3-corrector in $\mathcal{TC}'$ consisting of timed actions $\mathcal{TC}'5_i$ and $\mathcal{TC}'6_i$ with correction and witness predicates both equal to Q. Intuitively, $\mathcal{C}^1_{\mathcal{TC}'}$ ensures that when $\mathcal{TC}'$ is in a state outside the invariant, it reaches a state where both signals are red within 3 time units.

Likewise, for the property $\neg S_{\mathcal{TC}} \mapsto_{\leq 7} S_{\mathcal{TC}}$, let $\mathcal{C}^2_{\mathcal{TC}}$ be the strong 7-corrector consisting of timed actions $\mathcal{TC}'7_i$ and $\mathcal{TC}'8_i$ with witness and corrections predicates equal to $S_{\mathcal{TC}}$. In $\mathcal{C}^2_{\mathcal{TC}'}$, a z timer gets reset when the state of $\mathcal{TC}'$ is in $\neg S_{\mathcal{TC}} \wedge Q$ within 7 time units since the occurrence of a fault. Such a reset takes the traffic controller back to its invariant predicate $S_{\mathcal{TC}}$ where timed action $\mathcal{TC}1_i$ is enabled.

An alert reader notices that both $\mathcal{C}^1_{\mathcal{TC}'}$ and $\mathcal{C}^2_{\mathcal{TC}'}$ exhibit Zeno behavior when running individually. However, observe that $\mathcal{C}^2_{\mathcal{TC}'}$ provides new execution paths where $\mathcal{C}^1_{\mathcal{TC}'}$'s only choice is executing delay actions $\mathcal{TC}'6_i$. Moreover, when reaching the invariant predicate using $\mathcal{C}^2_{\mathcal{TC}'}$, timed and delay action of the intolerant program prevent the δ-corrector $\mathcal{C}^2_{\mathcal{TC}'}$ to exhibit a Zeno behavior.

10.3.4 The Necessity of Existence of Strong δ-Correctors in Hard-Masking Programs

We are now ready to prove that hard-masking programs contain hard-masking tolerant strong δ-correctors. Our strategy to accomplish our goal is as follows. First, in Claim 10.3.5, we show that *in the absence of faults*, if a program refines a specification within θ time units then it contains strong δ-correctors for some $\delta \in \mathbb{Z}_{\geq 0}$. Then, (*in the presence of faults*), using Claim 10.3.5, we show that if a hard-masking program $\mathcal{P}'$ is designed by reusing $\mathcal{P}$ to tolerate a set f of faults, $\mathcal{P}'$ contains a hard-masking tolerant strong δ-corrector. This is shown in Theorem 10.3.8.

Notation. For simplicity, we use the pseudo-arithmetic expressions to denote timing constraints over finite computations. For instance, $\overline{\sigma}_{\leq\delta}$, denotes a finite computation $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots (\sigma_n, \tau_n)$ that satisfies the timing constraint $\tau_n - \tau_0 \leq \delta$, where $\delta \in \mathbb{Z}_{\geq 0}$. We use S^* to denote a finite computation $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots (\sigma_n, \tau_n)$ st. $\sigma_i \models S$ for all i, $0 \leq i \leq n$. Thus, $(\mathit{true})^*_{\leq\infty}$ denotes an arbitrary finite computation with no specific time bound.

Definition 10.3.4 (becomes) Let $\mathcal{P}$ and $\mathcal{P}'$ be two real-time programs. We say that *$\mathcal{P}'$ becomes $\mathcal{P}$ within θ from T* iff $\mathcal{P}'$ refines $(\mathit{true})^*_{\leq\theta}\Gamma$ from T, where Γ is the set of all computations of $\mathcal{P}$. ∎

In Claim 10.3.5, we show that given a program $\mathcal{P}$, state predicate S, and specification SPEC, where $\mathcal{P}$ refines SPEC from S, if a program $\mathcal{P}'$ is designed st. it behaves like $\mathcal{P}$ within θ and, thus, has a suffix in SPEC, then $\mathcal{P}'$ is a strong δ-corrector of an invariant predicate of $\mathcal{P}$ for some $\delta \in \mathbb{Z}_{\geq 0}$. We prove this Claim by showing that $\mathcal{P}'$ itself refines the required strong δ-corrector specification.

Claim 10.3.5 *Let $\mathcal{P}$ and $\mathcal{P}'$ be real-time programs, Inv and S be nonempty state predicates, SPEC be a specification, and θ be a nonnegative integer.*
If

- *$\mathcal{P}$ refines SPEC from Inv,*

- *$\mathcal{P}'$ refines $\mathcal{P}$ from Inv, and*

- *$\mathcal{P}'$ becomes $\mathcal{P} \triangleright S$ within θ from S,*

then

- *there exists $\delta \in \mathbb{Z}_{\geq 0}$ st. $\mathcal{P}'$ is a strong δ-corrector of Inv.*

Proof. We prove this claim by, first, instantiating recovery time, whiteness predicate, and correction predicate as follows:

$\delta = \theta$,

$C = Inv$, and

$W = Inv \wedge \{\sigma \mid \sigma$ is a state of $\mathcal{P}'$ st. it is reached in some computation of $\mathcal{P}'$ starting from $S\}$.

The rest of the proof is to verify that by this instantiation, $\mathcal{P}'$ refines 'W strongly corrects C within δ' from S via showing Strong Convergence, Safeness, Progress, and Stability. To this end, we proceed as follows:

- (*Safeness*) By definition of W, in any state where W is true, Inv is true. Thus, Safeness is satisfied.

- (*Progress*) Since $\mathcal{P}'$ becomes $\mathcal{P} \triangleright Inv$ within θ from T, i.e., $\mathcal{P}'$ refines $(true)^*_{\leq\theta}(\mathcal{P} \triangleright Inv)$ from S, every computation of $\mathcal{P}'$ starting from S will reach a state where Inv is true within θ. By definition of W, W is true in this state as well. Thus, Progress is satisfied.

- (*Stability*) Since $\mathcal{P}'$ refines $\mathcal{P}$ from Inv, it follows that Inv is closed in $\mathcal{P}'$. Obviously, the second conjunct in W is closed in $\mathcal{P}'$ as well and, thus, W is closed in $\mathcal{P}'$. Hence, Stability is satisfied.

- (*Strong Convergence*) Since $\mathcal{P}'$ refines $(true)^*_{\leq\theta}(\mathcal{P} \triangleright Inv)$ from S, every computation of $\mathcal{P}'$ that originates from S reaches a state where Inv is true within θ. Moreover, $Inv(= C)$ is closed in $\mathcal{P}'$. Thus, since $\delta = \theta$, Strong Convergence within δ is satisfied. ∎

The next lemma generalizes Claim 10.3.5. In general, given a program $\mathcal{P}$ that refines $SPEC$ from S, $\mathcal{P}'$ may not behave like $\mathcal{P}$ from each state in S but only from a subset of S, say R. This may happen, for example, if $\mathcal{P}'$ contains additional variables and $\mathcal{P}'$ behaves like $\mathcal{P}$ only after the values of these additional variables are restored. Lemma 10.3.6 shows that in such a case, $\mathcal{P}'$ contains a hard-masking tolerant strong δ-corrector of an invariant predicate of $\mathcal{P}$. The strong δ-corrector is hard-masking in the sense that the correction predicate is preserved only after $\mathcal{P}'$ reaches a state where R is true.

Lemma 10.3.6 *Let $\mathcal{P}$ and $\mathcal{P}'$ be real-time programs, R, Inv, and S be nonempty state predicates where $R \Rightarrow Inv$, $SPEC$ be a specification, and θ be a nonnegative integer.*
If

- *$\mathcal{P}$ refines $SPEC$ from Inv,*
- *$\mathcal{P}'$ refines $\mathcal{P}$ from R, and*
- *$\mathcal{P}'$ becomes $(\mathcal{P} \triangleright R)$ within θ from S,*

then

- *there exists $\delta \in \mathbb{Z}_{\geq 0}$ st. $\mathcal{P}'$ is a hard-masking strong δ-corrector with recovery time θ of Inv.*

Proof. First, we instantiate correction and witness predicates as follows:

$C = Inv$, and
$W = R$.

We now show that there exists $\delta \in \mathbb{Z}_{\geq 0}$ st. $\mathcal{P}'$ refines the hard-masking tolerance specification with recovery time θ of 'W strongly corrects C within δ' from S. In particular, we first show that a computation of $\mathcal{P}'$ that starts from a state where S is true reaches a state where R is true within θ time units. Then, we show that starting from this state $\mathcal{P}'$ refines 'W strongly corrects C within δ'.

- For the first part, since $\mathcal{P}'$ becomes $\mathcal{P} \triangleright R$ within θ from S, it follows that $\mathcal{P}'$ reaches a state where R is true within θ.
- For the second part, we show that there exists $\delta \in \mathbb{Z}_{\geq 0}$ such that starting from S, $\mathcal{P}'$ satisfies Safeness, Progress, Stability, and Strong Convergence within δ. Thus, we proceed as follows:

– (*Safeness*) Since $R \Rightarrow Inv$ is trivially true, Safeness is satisfied by construction.

– (*Progress*) In a state where R is true, Inv also holds. Thus, Progress is satisfied.

– (*Stability*) Since $\mathcal{P}'$ refines $\mathcal{P}$ from R and R is closed in $\mathcal{P}'$, Stability is satisfied.

– (*Strong Convergence*) Finally, since in a computation starting from a state where R is true, Inv is immediately true at all states and $\mathcal{P}'$ is closed in Inv, by choosing $\delta = 0$ Strong Convergence is satisfied. ∎

We now illustrate the role of strong δ-correctors in hard-masking programs in the presence of faults. In particular, we use Claim 10.3.5 and Lemma 10.3.6 to show that if a hard-masking F-tolerant program $\mathcal{P}'$ with recovery time θ is designed by reusing $\mathcal{P}$ then there exists $\delta \in \mathbb{Z}_{\geq 0}$ st. $\mathcal{P}'$ contains a hard-masking F-tolerant strong δ-corrector with recovery time θ for an invariant predicate of $\mathcal{P}$. Notice that, since our goal is to identify components that provide *bounded-time* recovery in the presence of faults, there needs to be some bound on the number of occurrence of faults. In fact, it is straightforward to show that providing bounded-time recovery in the presence of unbounded occurrence of faults is generally impossible.

Assumption 10.3.7 Let $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$ be a program and F be a set of faults. Also, let $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots (\sigma_n, \tau_n)$ be a computation prefix in $\mathcal{P}[]F$ where $\sigma_0 \models Inv$ and $\sigma_i \not\models Inv$, $1 \leq i \leq n$. We assume that the number of occurrence of faults between states σ_1 and σ_n is at most k for some $k \in \mathbb{Z}_{\geq 0}$. ∎

Theorem 10.3.8 *Let $\mathcal{P}$ and $\mathcal{P}'$ be real-time programs, R and Inv be nonempty state predicates, $SPEC$ be a specification, and θ be a nonnegative integer.*
If

- *$\mathcal{P}$ refines $SPEC$ from Inv,*
- *$\mathcal{P}'$ is hard-masking F-tolerant for $SPEC$ from R, where $R \Rightarrow S$,*
- *$\mathcal{P}'$ refines $\mathcal{P}$ from R, and*
- *$\mathcal{P}'$ becomes $\mathcal{P} \triangleright R$ within θ from S, where S is an F-span of $\mathcal{P}'$,*

then

- *there exist δ and θ' in $\mathbb{Z}_{\geq 0}$ st. $\mathcal{P}'$ is a hard-masking F-tolerant strong δ-corrector with recovery time θ' of Inv.*

Proof. First, we construct a fault-span of $\mathcal{P}'$ for the rest of our proof. Let S_1 be the fault-span used to show that $\mathcal{P}'$ is hard-masking F-tolerant from R. Let $S_2 = S_1 \wedge S$. Since S_1 and S are F-spans, both are closed in $\mathcal{P}'[]F$. Hence S_2 is also closed in $\mathcal{P}'[]F$. Moreover, since $S_2 \Rightarrow S$, $\mathcal{P}'$ becomes $\mathcal{P} \triangleright R$ from S_2 as well.

We instantiate correction and witness predicates as follows:

$C = Inv$,
$W = R$.

We now show that there exist $\delta, \theta' \in \mathbb{Z}_{\geq 0}$ st. $\mathcal{P}'$ is hard-masking F-tolerant to 'W strongly corrects C within δ' with recovery time θ' from R. To this end, we first show that there exists $\delta \in \mathbb{Z}_{\geq 0}$ st. $\mathcal{P}'$ refines 'W strongly corrects C within δ' from R. Then, we show that there exists $\delta, \theta' \in \mathbb{Z}_{\geq 0}$ st. $\mathcal{P}'[]F$ refines the hard-masking tolerance specification of 'W strongly corrects C within δ' with recovery time θ' from S_2.

For the first part, from Lemma 10.3.6, we know that there exists δ st. $\mathcal{P}'$ refines hard-masking tolerance specification of 'W strongly corrects C within δ from R. Thus, the first part of proof is discharged. For the second part, we instantiate the recovery times as follows:

$\delta = \theta$, and
$\theta' = (k + 1).\theta$,

where k is the bound on the number of faults that may occur in a computation (cf. Assumption 10.3.7). Observe that, after occurrence of at most k faults, $\mathcal{P}'$ reaches a state in S_2 within at most $k.\theta$ time units. Moreover, from this state, $\mathcal{P}'$ refines 'W strongly corrects C within $\delta(=\theta)$' from S_2 (proof would be identical to proof of Lemma 10.3.6). Thus, after faults stop occurring, $\mathcal{P}'$ reaches a state where R is true. Therefore, $\mathcal{P}'$ is a hard-masking F-tolerant strong δ-corrector with recovery time $\theta' = (k+1).\theta$ to 'W strongly corrects C within δ' from R.

Remark. In the proof of Theorem 10.3.8, we showed that the recovery time that a strong δ-corrector provides in worst case depends on the maximum number of occurrence of faults. However, one can design correctors with better recovery time. For instance, in Chapter 8, we presented automated methods to synthesize one type of such correctors using state exploration and Dijkstra's shortest path algorithm. ∎

10.3.5 The Necessity of Existence of Weak δ-Correctors in Hard-Masking Programs

Just like the relation between recovery constraint and strong δ-correctors in hard masking programs, we show that if a hard-masking F-tolerant program $\mathcal{P}'$ is designed by reusing program $\mathcal{P}$ then $\mathcal{P}'$ contains a hard-masking tolerant weak δ-corrector for each stable bounded response property in $SPEC_{\overline{br}}$.

Theorem 10.3.9 *Let $\mathcal{P}$ and $\mathcal{P}'$ be real-time programs, R and Inv be nonempty state predicates where $R \Rightarrow Inv$, $SPEC$ be a specification, and θ be a nonnegative integer.*
If

- *$\mathcal{P}$ refines $SPEC$ from Inv,*
- *$\mathcal{P}'$ reuses $\mathcal{P}$ from R,*
- *$\mathcal{P}'$ is hard-masking F-tolerant to $SPEC$ from R*

then

- *($\forall i \mid 0 \leq i \leq m$: there exists $\delta \in \mathbb{Z}_{\geq 0}$ st. $\mathcal{P}'$ is a hard-masking tolerant weak δ-corrector for the response predicate of stable bounded response property $P_i \mapsto_{\leq \delta_i} Q_i$ of $SPEC_{\overline{br}}$).*

Proof.

Following Definition 7.1.6, $SPEC_{\overline{br}}$ consists of $m+1$ stable bounded response properties of the form $P_i \mapsto_{\leq \delta_i} Q_i$, where $0 \leq i \leq m$. In order to show the existence of hard-masking weak δ-correctors, for each stable bounded response property in $SPEC_{\overline{br}}$, we let

$C = Q_i$,
$W = Q_i \wedge \{\sigma \mid \sigma$ is a state of $\mathcal{P}'$ st. σ is reached in some computation of $\mathcal{P}'$ starting from $P_i\}$.

Now, we need to show that there exist $\delta, \theta' \in \mathbb{Z}_{\geq 0}$ st. $\mathcal{P}'$ is hard-masking F-tolerant to 'W weakly corrects C within δ' with recovery time θ' from R. By instantiating $\delta = \delta_i$, for each $i \in 0..m$, the rest of the proof strategy is identical to that of Theorem 10.3.8. The only point of difference is that for this theorem, we only need to show Weak Convergence, as $\mathcal{P}'$ does not have to be closed in each Q_i.

Chapter 11

Symbolic Synthesis of Distributed Fault-Tolerant Programs

The complexity of automated synthesis can be characterized in two parts. The first part has to deal with questions such as *which* recovery transitions/actions should be added, and *which* transitions/actions should be removed to prevent safety violation in the presence of faults. The second part has to deal with questions such as *how quickly* such recovery and safety violating transitions can be identified.

Observe that the solution to the first part is independent of issues such as representation of programs, faults, specifications etc. Hence, in previous chapters of this dissertation, we utilized explicit-state (enumerative) techniques to identify the complexity of revision. Explicit-state techniques are especially valuable in this context, as we can identify how different heuristics affect a given program, and thereby enable us to identify circumstances where they might be useful. Explicit-state techniques, however, are undesirable for the second part, as they suffer from state explosion problem and prevent one from synthesizing programs where the state space is large. In other words, although the polynomial-time complexity of the heuristics (e.g., in [KAC01]) allow us to deal with the problem of synthesis of distributed programs, which is known to be NP-complete, their explicit-state implementation is problematic with scaling up for large programs.

With this motivation, in this chapter, we focus on the second part of the problem to improve the time and space complexity of synthesis. Towards this end, we focus on symbolic synthesis (implicit-state) where programs, faults, specifications etc., are modeled using

Boolean formulae (represented by Bryant's Ordered Binary Decision Diagrams [Bry86]). Although symbolic techniques have been shown to be very successful in model checking [BCM+92] (e.g., in model checkers SMV and SAL), they have not been greatly used in the context of program synthesis and transformation in the literature. Thus, in this chapter, our goal is to evaluate how such symbolic synthesis can assist in reducing the time and space complexity, and thereby permit synthesis of large(r) programs.

Our contributions in this chapter are as follows.

1. We illustrate that our symbolic technique can significantly improve the performance of synthesis in terms of both time and space complexity. In particular, our analysis shows that the growth of the total synthesis time is *sublinear* in the state space. For example, in case of Byzantine agreement for five processes, the time spent for explicit-state synthesis was 15 minutes whereas the time spent for symbolic synthesis was less than a second.

2. Symbolic synthesis significantly assists in coping with the space complexity. For example, we could synthesize a solution for Byzantine agreement with 25 processes. The size of reachable states in this case is 10^{19}, whereas in our implementation the amount of memory used during synthesis was only 14.2 MB. To the best of our knowledge, this is the first instance that can deal with such large state space in the context of program synthesis.

3. We analyze the cost incurred in different tasks during synthesis. In particular, our analysis identifies several potential bottlenecks that need to be overcome, namely, (1) deadlock resolution, (2) computation of reachable states in the presence of faults, (3) checking whether a group of transitions violates the safety specification, (4) cycle detection, and (5) addition of recovery paths. We show that depending upon the structure of distributed programs, a combination of these bottlenecks may affect the performance of automated synthesis.

We would like to note that just as with model checking, this work does not imply that synthesis would be feasible for all programs where the size reachable of states is 10^{19}. However, this work does illustrate that large state space by itself is not an obstacle to permit efficient synthesis. Finally, while techniques such as symmetry reduction or other abstraction techniques have the potential to reduce the complexity of synthesis, with the use of such techniques, the actual state space of the given problem may not correspond to the state space encountered during synthesis. Since our goal in this work has been to focus on feasibility of dealing with large state space and its impact to the automated synthesis and transformation of distributed programs, we have chosen not to use such techniques.

The rest of this chapter is organized as follows. In Section 11.1, we present our symbolic heuristic for adding fault-tolerance to distributed programs. In Sections 11.2-11.6, we present our experimental results and analysis of the performance of our symbolic heuristics. Our analysis identifies potential bottlenecks of our methods based on the structure of the input program.

11.1 The Symbolic Synthesis Algorithm

In this section, we present our symbolic algorithm based on the heuristics developed in [KAC01]. In particular, the algorithm is formalized in terms of Boolean formulae which will later enable us to implement them using BDDs.

Algorithm sketch. The algorithm takes an input intolerant program, a safety specification, and a set of fault transitions and synthesizes a fault-tolerant program that satisfies the constraints of the Synthesis Problem 7.3.2. The algorithms consists of five steps. The first step is initialization, where we identify state and transition predicates from where execution of faults alone may violate the safety specification. The algorithm makes these state and transition predicates unreachable in order to guarantee that the synthesized fault-tolerant program does not violate the safety. In Step 2, we identify the fault-span by computing the state predicate reachable by program in the presence of faults, starting from the program invariant. In Step 3, the algorithm identifies and rules out transitions whose execution violates the safety specification. Then, in Step 4, we resolve deadlock states. Resolving deadlock states is crucial in the sense that existence of such states contributes in creation of finite computations that the input fault-intolerant program does not exhibit. Hence, in order to ensure that the synthesized fault-tolerant program satisfies liveness specification of the input program, deadlock states must be handled by either adding recovery paths or state elimination. Finally, in Step 5, we re-compute the invariant predicate so that it is closed in the final program. We repeat steps 2-3, 2-4, and 2-5 until a fixpoint is reached. The fixpoint computations are represented by three nested repeat-until loops in the algorithm. Thus, the algorithm terminates when no progress is possible in all the steps described above.

We now describe the algorithm Symbolic_Add_FT (cf. Algorithm 11.1) in detail:

- **Step 1: Initializations (Lines 1-3).** First, we compute the state predicate ms from where execution of faults alone violate the safety specification (Line 1). To this end, we start from state predicate where $Guard(F \wedge SPEC_{bt})$ is true (i.e., states from where faults directly violate the safety specification) and explore backward reachable states by applying fault transitions only. This is achieved by invoking the procedure BWReachStates as a black box. The first parameter of the procedure is the state predicate from where we start computing backward reachable states. The second parameter is the transition predicate applied for identifying reachable states. The procedure FWReachStates works in a similar fashion. The only difference is it explores forward reachable states. Symbolic implementation of these procedures has been studied extensively in automated verification techniques [BCM+92, McM93, BCL91]. Nonetheless, as we will illustrate in Sections 11.2-11.6, small changes in implementation may have tremendous impact on efficiency of state exploration with respect to different programs. This impact is often more dramatic in program synthesis than verification since reachability analysis procedures may be applied multiple times during the course of synthesizing a program.

  Since a program do not have control over the occurrence of faults, one has to ensure that the state predicate ms never becomes true in any program computation. Otherwise, faults alone may lead the program to a state where the safety is violated. Thus, we remove ms from the invariant of the fault-tolerant program (Line 2). We also compute the transition predicate mt whose transitions should not be executed by the fault-tolerant program. Initially, mt is equal to the union of $SPEC_{bt}$ and transitions that start from any arbitrary state and end in ms (Line 3). Since the fault-tolerant program is not supposed to reach a state in ms, we allow the transitions that *originate* in *ms* to be in the fault-tolerant program (Line 3). Notice that in Algorithm Add_Symbolic_FT and its sub-procedures, any addition or removal of a transition predicate has to be applied along with its group predicate due to the issue of read restrictions. Thus, issues with regard to distributed processes are all handled in computing group predicates. In fact, when allowing the program to execute transitions that originate in *ms* automatically includes other transitions that may originate outside *ms*. This inclusion increases the level of non-determinism and, hence, diversity of existence of program transitions at reachable states. Such diversity often increases chances for successful synthesis as well. Observe that although a state predicate, in Line 3, one can interpret *ms* as a transition predicate that starts in *ms* and ends in *true*.

- **Step 2: Re-computing the fault-span (Lines 9-11).** After initializations, we start re-computing the invariant predicate, program transition predicate, and fault-span of the fault-tolerant program in three nested loops, respectively. Each loop resembles a fixpoint calculation. We start with the most inner loop, where we re-compute the fault-span. The reason for re-computing the fault-span is due to the fact that in other steps of our algorithm, we add and remove transitions that originate in the fault-span. Hence, after such addition or removal, new states may become reachable and some states may become unreachable. Thus, re-computation is needed to update the fault-span.

  Let the initial fault-span S_1 be equal to the invariant Inv_1 (Line 7). We re-compute the fault-span by starting exploration from states where the invariant Inv_1 is true and applying the program transition predicate in the presence of faults (i.e., $T_1 \vee F$). To this end, we invoke the procedure FWReachStates (Line 10). After recomputing the fault-span, we remove the state predicate *fte* from the new fault-span S_1 (Line 11). This state predicate is identified later in Step 4 where we resolve deadlock states. For now, *fte* contains states failed to eliminate during deadlock resolution.

- **Step 3: Identifying and removing unsafe transitions (Lines 12-13).** We first identify unsafe transitions. Suppose there exists a state predicate which is a subset of

Algorithm 11.1 Symbolic_Add_FT

Input: program transition predicate $T_{\mathcal{P}}$, invariant predicate $Inv_{\mathcal{P}}$, fault transitions F, and bad transition predicate $SPEC_{bt}$.

Output: program transition predicate $T_prime\mathcal{P}$ and invariant predicate $Inv_{\mathcal{P}'}$.

  // *initializations*
1: $ms \ := \ \text{BWReachStates}(Guard(F \wedge SPEC_{bt}),\ F)$;
2: $Inv_1 \ := \ Inv_{\mathcal{P}} - ms$;
3: $mt \ := \ (\langle ms \rangle' \ \vee \ SPEC_{bt}) \ \wedge \ Group(\neg ms)$;

4: **repeat** $\{$*recomputing the invariant predicate*$\}$
5:   $Inv_2 \ := \ Inv_1$;
6:   **repeat** $\{$*recomputing the transition predicate*$\}$
7:     $S_1,\ T_2 \ := \ Inv_1,\ T_1$;
8:     **repeat** $\{$*recomputing the fault-span*$\}$
9:       $S_2 \ := \ S_1$;
10:       $S_1 \ := \ \text{FWReachStates}(Inv_1,\ T_1 \vee F)$;
11:       $S_1 \ := \ S_1 - fte$;
12:       $mt \ := \ mt \wedge S_1$;
13:       $T_1 \ := \ T_1 \ - \ Group(T_1 \wedge \ \neg mt)$; $\{$*removing unsafe transitions*$\}$
14:     **until** $(S_1 \ = \ S_2)$;
    $\{$*Resolving deadlock states through adding recovery or state elimination*$\}$
15:     $ds \ := \ S_1 \wedge \neg Guard(T_1)$;
16:     $T_1 \ := \ T_1 \ \vee \ \text{AddRecovery}(ds,\ Inv_1,\ S_1,\ mt)$;
17:     $ds \ := \ S_1 \wedge \neg Guard(T_1)$;
18:     $T_1,\ fte \ := \ \text{Eliminate}(ds,\ T_1,\ Inv_1,\ S_1,\ F,\ false, false)$;
19:   **until** $(T_1 \ = \ T_2)$;
20:   $T_1,\ Inv_1 \ := \ \text{ConstructInvariant}(T_1,\ Inv_1,\ fte)$;
21: **until** $(Inv_1 \ = \ Inv_2)$;
22: $Inv_{\mathcal{P}'},\ T_prime\mathcal{P} \ := \ Inv_1,\ T_1$;
23: **return** $Inv_{\mathcal{P}'},\ T_prime\mathcal{P}$;

$Guard(mt)$, but it is unreachable by transitions in $T_1 \vee F$ starting from the invariant. In other words, it does not intersect with the fault-span. Since this state predicate is unreachable by computations of the program even in the presence of faults, we do not consider them as unsafe transitions and we let the transitions that originate in that state predicate be in the fault-tolerant program (Line 12). In other words, the necessary condition for a transition to be unsafe is its source state has to be in the fault-span. Otherwise, the transition can exist in the fault-tolerant program transition predicate, even if it is in $SPEC_{bt}$. The reason for not including such transitions in mt is due to the fact that their corresponding group predicates are also included in the synthesized program transition predicate which in turn adds to non-determinism and diversity of the program.

The fault-tolerant program transition predicate is computed based on the following principle: a transition can be included if it is not unsafe. Thus, once we identify unsafe transitions (i.e., transition predicate mt), we rule them out from the program transition predicate (Line 13). Note that a transition can be included in the fault-tolerant program if its entire corresponding group predicate can be included.

- **Step 4: Resolving deadlock states (Lines 15-18).** Since the algorithm may remove some transitions from the input program, some states may have no outgoing transitions due to this removal. Consequently, computations that reaches such states deadlock. Thus, when the execution of the algorithm reaches a fixpoint in the inner loop, we identify *deadlock states* inside the fault-span. We cure such deadlock states in two ways. We either add a safe *recovery* path from a deadlock state to the program invariant, or (if recovery is not possible) we *eliminate* the deadlock state. By state elimination, we mean making the state unreachable. The deadlock resolution mechanisms are implemented in two procedures AddRecovery and Eliminate (cf. Procedures 11.2 and 11.3), respectively. We now describe these mechanisms in detail. First, the algorithm identifies the deadlock state predicate ds (Line 15). Intuitively, a state in the fault-span is deadlocked if the guard of T_1 is false in that state.

Procedure 11.2 AddRecovery

Input: deadlock states ds, invariant Inv, fault-span S, and transition predicate mt.
Output: recovery transition predicate rec.

```
lyr, rec := Inv, false;
repeat
   rt := Group(ds ∧ ⟨lyr⟩′);
   rt := rt − Group(rt ∧ mt);
   if DetectCycles(T ∨ rec ∨ rt, S) then
      rt := false;
   end if
   rec := rec ∨ rt;
   lyr := Guard(ds ∧ rt);
until (lyr = false);
return rec;
```

  - *(Adding safe recovery paths)* The Procedure AddRecovery (cf. Procedure 11.2) takes a deadlock state predicate *ds*, invariant predicate *Inv*, fault-span S, and unsafe transition predicate *mt* as input. It returns a transition predicate which contains new recovery transitions as output. We add recovery paths in an iteratively layered fashion. Let the first layer be the program invariant, i.e., $lyr = Inv_1$ (Line 1). Also, let *rt* be the transition predicate that originates from the deadlock state predicate *ds* and ends in *lyr* (Line 3). Since we require the fault-tolerant program to satisfy safety during recovery, *rt* must be disjoint from *mt*. Thus, we check whether the group predicate of *rt* maintains the safety specification (Line 4). If so, we check whether addition of *rt* to the program transition predicate creates a cycle that is entirely in the fault-span. Existence of such a cycle obviously prevents the program to always recover to the invariant predicate within a finite number of steps. To this end, we invoke the procedure DetectCycles with parameters $T \vee rec \vee rt$ and S. If there is indeed a cycle in the fault-span created due to addition of *rt* to the program transition predicate, we do not add the new recovery transitions *rt* to the *rec* which is the final set of recovery transitions. Otherwise, it is safe to add *rt* to *rec* (Line 8). There exist several symbolic approaches in the literature for detecting cycles [FFK$^+$01]. We, in particular, incorporate the approach introduced by Emerson and Lei [EL86].

In the next iteration, we let *lyr* be the state predicate from where one-step safe recovery is possible (Line 9). We continue adding recovery steps until no new recovery transition is added.

– *(Eliminating deadlock states)* Now, if safe recovery is not possible from a deadlock state predicate, we choose to eliminate it. By state elimination we mean making that state unreachable. The recursive Procedure Eliminate takes a deadlock state predicate *ds*, program transition predicate T, invariant predicate *Inv*, fault-span S, fault transitions F, a predicate *vds* of deadlock states visited while eliminating, and a predicate *fte* of deadlock states failed to eliminate as input. It returns a program transition predicate $T^{'}$, visited deadlock states vds', and states fte' failed to eliminate as output.

  The Procedure Eliminate works as follows. First, if all deadlock states in *ds* has already been considered for elimination, the procedure returns immediately

Procedure 11.3 Eliminate

Input: deadlock states ds, transition predicate T, invariant Inv, fault-span S, fault transitions F, visited deadlock states vds, and predicate fte failed to eliminate.

Output: revised program transition predicate T, visited deadlock states vds, and predicate fte failed to eliminate, where states in ds become unreachable.

1: $ds \ := \ ds - vds;$
2: **if** $(ds = false)$ **then**
3:    $T^{'}, \ vds', \ fte' \ := \ T, \ vds, \ fte;$
4:    **return** $T^{'}, \ vds', \ fte';$
5: **end if**
6: $vds \ := \ vds \ \vee \ ds;$

7: $tmp \ := \ (S - Inv) \ \wedge T \ \wedge \langle ds \rangle';$ {*eliminating states in ds*}
8: $T \ := \ T - Group(tmp);$

9: $fs \ := \ Guard(S \ \wedge \ F \ \wedge \ \langle ds \rangle');$ {*eliminating source states of incoming fault transitions*}
10: $fte \ := \ fte \ \vee \ \langle fs \wedge F \rangle'';$
11: $OffendingStates \ := OffendingStates \ \vee \ (fs \wedge Inv);$
12: **if** $(fs \neq false)$ **then**
13:    $T, \ vds, \ fte \ := \ \text{Eliminate}(fs - Inv, \ T, \ Inv, \ S, \ F, \ vds, \ fte);$
14: **end if**

15: $nds \ := \ Guard(S \wedge Group(tmp) \wedge \neg T);$ {*testing whether removal of incoming program transitions creates new deadlock states*}
16: $T \ := \ T \ \vee \ (Group(tmp) \wedge nds);$
17: $fte \ := \ fte \ \vee \ ((nds \wedge \langle Group(tmp) \rangle'') \ - Inv);$
18: $T^{'}, \ vds', \ fte' \ := \ \text{Eliminate}(nds, \ T, \ Inv, \ S, \ F, \ vds, \ fte);$

19: **return** $T^{'}, \ vds', \ fte';$

(Lines 1-5). Otherwise, we add ds to the predicate vds of states already visited (Line 6). There are potentially two ways to reach the states in ds: (1) by program transitions, and (2) by fault transitions. If ds is reachable by some fault transitions, we need to backtrack to the source state predicate and make that predicate unreachable. If ds is reachable by program transitions, we remove those transitions and their corresponding group predicates from T, provided no new deadlock states are introduced. Thus, we proceed as follows. If ds is reachable by some program transitions, we temporarily remove such transitions, say tmp, with the hope that this removal makes ds unreachable (Lines 7-8). Unlike program transitions, since the program does not have control over the occurrence of faults, if there exist states in *ds* that are reachable by fault transitions from another state predicate, say *fs* (Line 9), then we backtrack and eliminate *fs*. Thus, we mark the states reachable by fault transitions from *fs* as *failed to eliminate* (Line 10). Elimination of states in *fs* is based on the following principle: we do not eliminate states in the invariant predicate. Thus, if *fs* intersects with the invariant, we mark the intersection as *OffendingStates* and deal with them when we re-compute the invariant predicate in Step 5. At this point, we invoke Eliminate recursively with parameter $fs - Inv$ (Line 13).

Once we deal with incoming fault transitions, we ensure that removal of $Group(tmp)$ does not introduce new deadlock states to the program. If this is the case, the predicate *nds* (computed in Line 15) is not equal to false. Thus, we take the following steps: We

1. add the transitions originating from *nds* back to the program (Line 16),
2. mark the states in *nds* as failed to eliminate (Line 17), and
3. attempt to ensure that *nds* is never reached by invoking Eliminate with parameter *nds* recursively (Line 18).

As mentioned earlier the Algorithm Symbolic_Add_FT exploits the above mechanisms to deal with deadlock states. As can be seen, first it invokes the Procedure AddRecovery on the existing deadlock states (Line 16). Then, if AddRecovery fails to resolve some deadlock states, we make them unreachable by invoking the Procedure Eliminate (Line 18). Finally, notice that in Line 11 of the algorithm, we exclude state predicate *fte* from the re-computed fault-span. This is due to the face that these states have to eventually become unreachable and if we include them in the fault-span, in the next iteration of resolving deadlock states, they will be unnecessarily re-considered for elimination.

- **Step 5: Re-computing the invariant (Line 20).** Once we reach a fixpoint after re-computing fault-span and program transition predicate, we re-compute the invariant by invoking the Procedure ConstructInvariant (Line 20) due to possible existence of offending states. Recall that a computation, say $\overline{\sigma} = \sigma_0 \rightarrow \sigma_1 \rightarrow \cdots$, that starts from an offending state may reach a state that was considered for elimination. In

Procedure 11.4 ConstrucInvariant

Input: invariant predicate Inv, program transition predicate T, and offending states $OffendingStates$.

Output: revised invariant predicate Inv' and program transition predicate $T^{'}$.

1: **while** $(OffendingStates \neq false)$ **do**
2: $\quad Inv \ := \ Inv \ - \ OffendingStates;$
3: $\quad tmp \ := \ T \wedge Inv \wedge \langle \neg Inv \rangle';$
4: $\quad T \ := T \ - \ Group(tmp);$
5: $\quad OffendingStates \ := \ \langle tmp \rangle'';$
6: **end while**

7: $Inv', \, T^{'} \ := \ Inv, \, T;$
8: **return** $Inv', \, T^{'};$

particular, since the first transition of $\overline{\sigma}$ is a fault transition, σ_0 which is an offending state has to become unreachable. To this end, we first remove offending states from the invariant (Line 2). Then, due to this removal, we need to ensure that the invariant predicate is closed in T. Thus, we remove the transition predicate that violates the closure of T (Lines 3 and 4). We continue these step until a fixpoint is reached in the sense that no offending states exist and Inv is closed in T.

The algorithm keeps repeating steps 1-5 until the three fixpoints are reached. At the end of each fixpoint computation, we verify the correctness of conditions of the Synthesis Problem 7.3.2. Hence, when the algorithm terminates, we are guaranteed the solution is sound.

Regarding the structure of the output program in terms of guarded commands, one can notice that the output of our algorithm typically consists of three types of actions as compared to the input intolerant program. These action type are the following:

1. *Unchanged actions.* These actions identically exist in both tolerant and intolerant programs.

2. *Strengthened actions.* Transitions corresponding to these actions exist in the input program. However, the guard of the corresponding actions are stronger than their counterpart actions in the input program. This is due to the fact that the transformed program makes unsafe states and some deadlock states unreachable.

3. *Recovery actions.* These action do not exist in the input program at all. Recovery

actions are added to the input program in order to resolve some deadlock states that are reachable by the program in the presence of faults.

In the next five sections, we present the experimental results of implementation of the Algorithm Symbolic_Add_FT and its procedures in our tool SYCRAFT [BK08c]. Our case studies include three classic examples in the literature of distributed fault-tolerant computing, namely, the Byzantine agreement problem [LSP82], Byzantine agreement with fail-stop faults, and the token ring [AK98a] problem. We also present experimental results on addition of fault-tolerance to a bulk data dissemination protocol in wireless sensor networks, known as Infuse [KA06]. As mentioned in the introduction, our case studies address a wide variety of structural issues and obstacles that can potentially affect the efficiency of our algorithm. In all case studies, we find a considerable improvement in both time and space complexity as compared to the existing approaches.

11.2 Case Study 1: Byzantine Agreement

Throughout this section and Sections 11.3, 11.4, 11.5, 11.6 , all experiments are run on a dedicated PC with a 2.2GHz AMD Opteron processor and 1.2GB RAM. The BDD representation of the Boolean formulae is implemented using the Glu/CUDD package [Som]. We analyze all the experiments in terms of *time* and *space*. From the time perspective, we consider total synthesis time, time spent for resolving deadlock states (including addition of safe recovery and state elimination), cycle detection, and computing the fault-span. From the space perspective, we consider the number of states reachable by each distributed program in the presence of faults (i.e., the size of fault-span) and the actual memory usage. Our first case study is the continuation of our running example, the Byzantine agreement program.

Example (cont'd). The output of our algorithm with respect to program $\mathcal{BA}$ is program $\mathcal{BA}'$ which tolerates the Byzantine faults identified in Section 7.2.3 in the sense that $\mathcal{BA}'$ never violates its specification and it does not deadlock when faults occur. We note that our synthesized program is identical to the canonical version of Byzantine agreement program manually designed in [LSP82]. The actions of the synthesized program for a non-general process j are as follows:

$$
\begin{array}{lll}
\mathcal{BA}'1_j & :: & d.j = \perp \;\wedge\; f.j = \mathit{false} \\
& & \qquad \longrightarrow \quad d.j \;:=\; d.g; \\
\mathcal{BA}'2_j & :: & d.j \neq \perp \;\wedge\; f.j = \mathit{false} \;\wedge\; (d.k{=}\perp \;\vee\; d.k = d.j) \;\wedge \\
& & (d.l{=}\perp \;\vee d.l = d.j) \;\wedge (d.k \neq \perp \vee d.l \neq \perp) \\
& & \qquad \longrightarrow \quad f.j \;:=\; \mathit{true}; \\
\mathcal{BA}'3_j & :: & d.j = 1 \;\wedge\; d.k = 0 \;\wedge\; d.l = 0 \;\wedge\; f.j = \mathit{false} \\
& & \qquad \longrightarrow \quad d.j,\; f.j \;:=\; 0,\; \mathit{false}|\mathit{true}; \\
\mathcal{BA}'4_j & :: & d.j = 0 \;\wedge\; d.k = 1 \;\wedge\; d.l = 1 \;\wedge\; f.j = \mathit{false} \\
& & \qquad \longrightarrow \quad d.j,\; f.j \;:=\; 1,\; \mathit{false}|\mathit{true}; \\
\mathcal{BA}'5_j & :: & d.j \neq \perp \;\wedge\; f.j{=}\mathit{false} \wedge \\
& & ((d.j{=}d.k \wedge d.j{\neq}\, d.l) \;\vee\; (d.j{=}d.l \wedge d.j{\neq}\, d.k)) \\
& & \qquad \longrightarrow \quad f.j \;:=\; \mathit{true};
\end{array}
$$

Notice that action $\mathcal{BA}'1$ is unchanged, actions $\mathcal{BA}'3$ and $\mathcal{BA}'4$ are recovery actions, and actions $\mathcal{BA}'2$ and $\mathcal{BA}'5$ are strengthened actions.

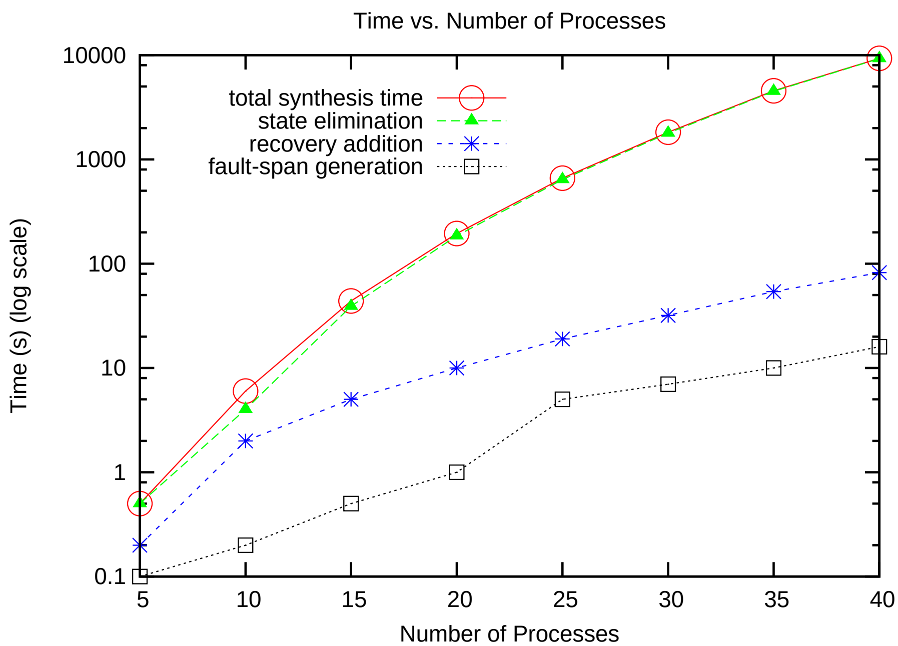
We now present the results of our experiments with respect to the Byzantine agreement program. Figure 11.1-a shows actual memory usage in megabytes and time needed to accomplish each subtask of the algorithm in seconds in terms of the number of non-general processes ($\mathcal{BA}^i$ denotes Byzantine agreement program with i non-general processes). Figure 11.1-b compares the total synthesis time and the time spent to complete subtasks of the algorithm graphically. Notice that the x-axis is in logarithmic scale. The number of processes synthesized in our experiments ranges over 5 to 40. Although it is feasible to synthesize programs with more number of processes in a reasonable amount of time, the trend of the graph with maximum 40 processes is sufficiently clear to make sound judgments. We now analyze the results.

Total synthesis time

First, observe that it takes less than one second to synthesize 5 non-general processes. It is noteworthy to mention that an enumerative implementation of similar heuristics [KAC01, EKAar] requires 900 seconds to synthesize the same number of processes. Moreover, this enumerative implementation can not handle more than 5 processes due to the state explosion

	Space		*Time(s)*			
	reachable states	**memory (MB)**	**deadlock resolution**		**fault-span generation**	**total**
			Eliminate	Recovery		
$\mathcal{BA}^5$	10^4	3.5	< 1	< 1	< 1	< 1
$\mathcal{BA}^{10}$	10^9	6.2	4	2	< 1	6
$\mathcal{BA}^{15}$	10^{12}	11.5	39	5	< 1	45
$\mathcal{BA}^{20}$	10^{15}	13.68	185	10	1	199
$\mathcal{BA}^{25}$	10^{19}	14.2	642	19	5	669
$\mathcal{BA}^{30}$	10^{22}	15.2	1791	32	7	1836
$\mathcal{BA}^{35}$	10^{26}	15.6	4492	54	10	4565
$\mathcal{BA}^{40}$	10^{30}	16.5	9253	82	16	9366

(a)



(b)

Figure 11.1: Experimental results for algorithm Symbolic_Add_FT and the Byzantine agreement problem.

problem. To the contrary, our symbolic approach is cable of synthesizing 40 processes (10^{30} reachable states and beyond) in a reasonable amount of time, which is obviously a significant improvement. Note that the size of state space of $\mathcal{BA}^{40}$ is 10^{28} times larger than the size state space of $\mathcal{BA}^5$.

More importantly, Figure 11.1-b shows that the growth rate of total time spent to synthesize Byzantine agreement is *sublinear* to the size of reachable states, left alone the size of entire state space. In particular, our analysis shows that the fraction $\frac{Time}{ReachableStates^{0.13}}$ remains constant as the number of non-general processes grows. Sublinearity of total synthesis time to the size of state space is important in the sense that the exponential blow-up of state space does not affect the time complexity of our synthesis algorithm. More precisely, the size of reachable states is not an obstacle by itself in order to synthesize distributed programs.

Fault-Span Generation

As can be seen in Figure 11.1-b, the generation of fault-span is fairly fast in case of the Byzantine agreement program. This is mainly due to the following contributing factors:

1. The state space of the program is partially reachable. To illustrate the issue of size of reachable states let us consider the Byzantine agreement program with i processes. Since we represent the decision value of each processes by two Boolean variables, as the size of their respective domain is 3, each non-general process has Boolean 4 variables. Also, the general has 2 variables. Hence, the program has $4i + 2$ Boolean variables in total and the size of state space is 2^{4i+2}. In order to compute the size of reachable states approximately, observe that non-general processes are either undecided (i.e., $d.j = \perp$), or they are decided (i.e., $d.j = 0|1$) and their decision is either finalized or not yet finalized (i.e., $f.j = \mathit{false}|\mathit{true}$). Hence, each non-general can have 5 different combinations. Furthermore, the general can have either decision value (i.e., $d.g = 0|1$) and be Byzantine or non-Byzantine (i.e., $b.g = \mathit{false}|\mathit{true}$). Hence, the size of reachable states is at least $5^i * 4$. Thus, the size of reachable states is considerably less than the size of entire explicit state space. For instance, in case of $\mathcal{BA}^{40}$, the size of state space is 10^{45}, while only 10^{30} states are reachable.

2. The diameter of the state-transition graph of the program is not long and, hence, shallow reachability is possible. For instance, in case of $\mathcal{BA}^{30}$, the fault-span can be computed by only 32 rounds of frontier generation.

These reasons significantly affect the efficiency of fault-span generation. It is important to notice that the above factors may completely be the opposite in distributed programs with different structures. As a matter of fact, our experiments with respect to the token ring problem (see Section 11.5) validates this statement.

Deadlock resolution

Figure 11.1 also shows the time spent to resolve deadlock states for different number of processes. As mentioned earlier, deadlock resolution is crucial in order for the output program to meet liveness properties. We note that deadlock resolution (as defined in Section 11.1) is a problem that uniquely exists in the context of program synthesis and transformation and, hence, has not been addressed by the model checking community. In fact, there is very little experimental analysis with regard to synthesis of liveness properties in the literature. Since deadlock resolution is achieved through adding recovery paths and state elimination, we present the time spent in each subtask for a more thorough analysis.

Addition of safe recovery is the first attempt that the algorithm makes in order to resolve deadlock states. The results of our experiments in case of Byzantine agreement shows that this mechanism is not costly as compared to the total synthesis time (see Figure 11.1-b). This is solely due to the structure of the fault-span of $\mathcal{BA}$ where the size of invariant and safety specification are not barriers in efficient addition of safe recovery paths.

As can be seen in Figure 11.1-b, the graph of total synthesis time is almost identical to the graph of state elimination time. In fact, in the range of 5-40 processes, in average, 96% of the total synthesis time is spent to resolve deadlock states through state elimination. In other words, only 4% of the total synthesis time is spent to re-compute the fault-span, checking the safety of group predicates, computing recovery paths, and re-computing the program invariant. This is basically due to the fact that state elimination can potentially involve many backtracking steps. As a matter of fact, this is exactly what is happening in Byzantine agreement. For instance, in case of $\mathcal{BA}^5$, the Procedure Eliminate needs to be called 26 times recursively. Thus, state elimination can potentially be a serious stumbling block in efficiency of synthesis algorithms. We note that the existence and diversity of deadlock states directly depends on the structure of the given program. In Section 11.5, we show that in case of token ring, for instance, deadlock resolution is not a crucial issue.

Finally, the numbers in Figure 11.1-a show that if we were not required to resolve deadlocks states, identifying a solution to the problem can be accomplished considerably faster. Although such a solution is not a sensible *masking* fault-tolerant program, it is a correct *failsafe* fault-tolerant program. A failsafe program is one that is required to merely satisfy its safety specification and not necessarily its liveness specification in the presence of faults. Consequently, there is no need to resolve deadlock states in order to synthesize a failsafe solution. Thus, one can synthesize a failsafe solution to $\mathcal{BA}^{40}$ in 22 seconds. This result is significantly better than the DiConic approach in [Ebn07] where synthesis of a failsafe version of $\mathcal{BA}^{40}$ requires 353 seconds using a cluster of workstations.

Memory usage

Figure 11.1-a also shows the amount of virtual memory that the Algorithm Symbolic_Add_FT requires (in MB) for different number of non-general processes. As can be seen, the amount of memory that the algorithm requires to synthesize 40 processes (16.5 MB) is not considerably greater than the amount of memory required to synthesize 5 processes (3.5 KB). The insignificant growth trend of memory usage is more appreciable when it is compared to the growth of number of reachable states in case of 5 and 40 processes. Low memory usage of our algorithm with respect to this case study is clearly due to efficient representation of Boolean formulae by BDDs.

The issue of variable ordering

The key reason to efficient encoding of a Boolean formula in a BDD is to identify an appropriate order of variables when constructing the BDD [Bry86]. In our implementation, we order the variables based on the following two principles, regardless of the structure of the given fault-intolerant program:

1. Each primed variable is always ordered immediately after its corresponding unprimed variable, and

2. Variables of each process are ordered subsequently one after another.

For instance, in Byzantine agreement program, the order of variables is as follows:

$$d.j < d'.j < f.j < f'.j < b.j < b'.j < d.k < d'.k < f.k < f'.k < b.k < b'.k < \cdots.$$

The first principle is a rule of thumb in existing symbolic model checkers as well. This is due to the fact that a transition often updates a subset of program variables, say U, and leave the rest unchanged. Hence, for each variable v where $v \notin U$, $v = v'$ must hold in the BDD that encodes the transition. Therefore, in order to reduce the number of nodes in the BDD, it is more efficient to order each primed variable immediately right after its corresponding unprimed variable.

The second principle reduces the number of nodes in BDDs that encode group predicates. Recall that the value all readable variables in source state of all transitions in a group predicate are equal. The same premise holds for target states. Thus, it is beneficial to order readable variables of a process subsequently. As a concrete example, an implementation of Symbolic_Add_FT that does not apply the second principle requires one minute to synthesize $\mathcal{BA}^{10}$, which is 10 times slower than the case where the second principle is applied.

11.3 Case Study 2: Exploiting Human Knowledge to Assist Synthesis Algorithms

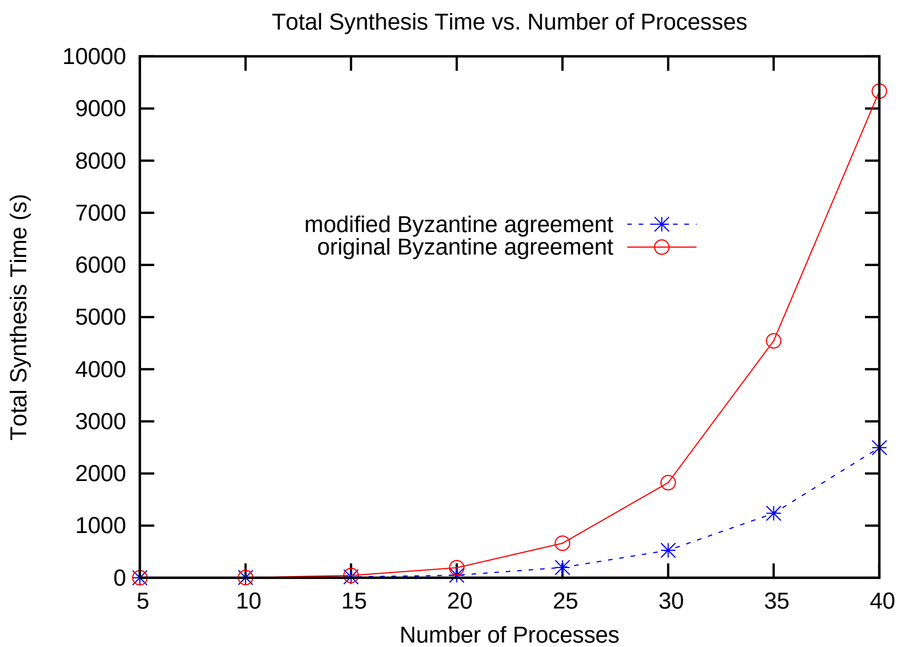
Our experiments on the Byzantine agreement problems clearly exhibited the most serious bottleneck, namely, state elimination. One way to remedy this problem is by making states that have to considered for elimination unreachable. This can be achieved through labeling transitions that can potentially lead a computation to reach such states as bad transitions in the safety specification. In order to analyze state elimination in the Byzantine agreement program, we we take a closer look at the state-transition graph of the $\mathcal{BA}$.

Let the sequence $\langle x_1, x_2, x_3, x_4 \rangle$ denote the set of states with respect to decision value of processes, i.e., $x_1 = d.g$, $x_2 = d.j$, $x_3 = d.k$, and $x_4 = d.l$. In this notation, an overlined (respectively, underlined) d-value shows that the corresponding process has finalized its decision (respectively, is Byzantine). Now consider the following scenario: Starting from a state s_0 in $\langle 1, \perp, \perp, 1 \rangle$, where the general and process l agree on decision 1 and processes j and k are undecided, the program may reach the following sequence of states due to the occurrence of faults (denoted $\dashrightarrow$) and execution of program actions (denoted $\rightarrow$): $\langle 1, \perp, \perp, 1 \rangle \rightarrow \langle 1, \perp, \perp, \overline{1} \rangle \dashrightarrow \langle \underline{1}, \perp, \perp, \overline{1} \rangle \dashrightarrow \langle \underline{0}, \perp, \perp, \overline{1} \rangle \rightarrow \langle \underline{0}, 0, \perp, \overline{1} \rangle \rightarrow \langle \underline{0}, 0, 0, \overline{1} \rangle$. Let s_1 be a state in $\langle \underline{0}, 0, 0, \overline{1} \rangle$, where non-general processes j and k agree with the Byzantine general on decision 0, but process l has finalized its decision on 1. Observe that s_1 is a deadlock state and since process l has finalized its decision, we cannot resolve s_1 by adding safe recovery. Thus, s_1 has to be eliminated.

One way to make s_1 and symmetrically equal states unreachable is by constraining a non-general process such that it is allowed to finalize its decision only when there does not

	Space		*Time(s)*			
	reachable states	**memory (MB)**	**deadlock resolution** Eliminate	**deadlock resolution** Recovery	**fault-span generation**	**total**
$\mathcal{BA}^5$	10^4	3	< 1	< 1	< 1	< 1
$\mathcal{BA}^{10}$	10^9	6	0	2	< 1	3
$\mathcal{BA}^{15}$	10^{12}	14.5	0	14	1	17
$\mathcal{BA}^{20}$	10^{15}	18	0	63	1	67
$\mathcal{BA}^{25}$	10^{19}	24	0	188	1	199
$\mathcal{BA}^{30}$	10^{22}	31	0	506	4	526
$\mathcal{BA}^{35}$	10^{26}	44	0	1203	7	1237
$\mathcal{BA}^{40}$	10^{30}	64	0	2428	25	2496

(a)



(b)

Figure 11.2: Experimental results for modified Byzantine agreement problem.

exist two ore more undecided non-generals. In this case, the the process cannot reach a state such as s_1. Thus, we modify the safety specification of $\mathcal{BA}$ (i.e., the bad transition predicate $SPEC_{bt_{\mathcal{BA}}}$ introduced in Subsection 2.2.1) as follows:

$$SPEC_{bt_{\mathcal{BA}}} \;=\; SPEC_{bt_{\mathcal{BA}}} \;\vee\; \exists p :: \neg b'.p \;\wedge\; f'.p \;\wedge\; (\exists q, r :: d.q \;=\; d.r \;=\; \bot),$$

where p, q, and r range over non-general processes.

Figure 11.2 shows the result of our experiments for Byzantine agreement with the above modification in the safety specification. One can make the following observations from this figure:

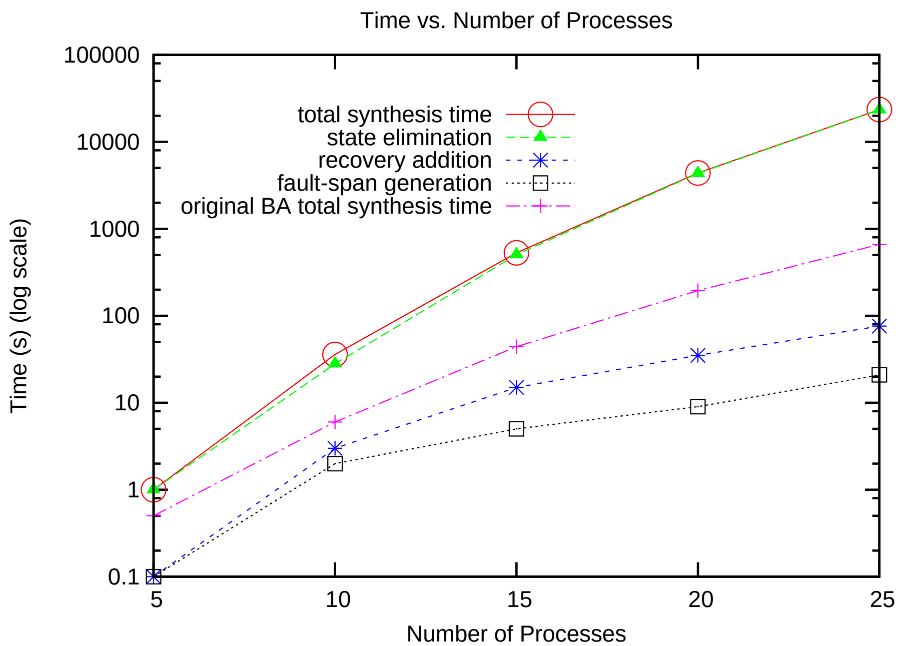
- Figure 11.2-a shows that no time is spent for state elimination. This is obviously due to non-existence of deadlock states from where safe recovery is not possible.
- Figure 11.2-b compares total synthesis time of the the original and modified versions of Byzantine agreement. One can obviously see a considerable improvement. Precisely, in average, the modified version can be synthesized 4 times faster than the original version for the range of 5 to 40 processes. It is expected that this factor becomes larger as the number of processes increases.
- Although constraining a non-general process casts away states that have to be eliminated, it affects the performance of addition of safe recovery. Our analysis shows this is mainly due to enlarging the transition predicate $SPEC_{bt_{\mathcal{BA}}}$ and as a result its corresponding BDD. Recall that recovery paths are not allowed to violate the safety specification and, hence, the large size of BDD that encodes $SPEC_{bt_{\mathcal{BA}}}$ affects the performance of addition of recovery. Nonetheless, this cost does not diminish the improvement of total synthesis time.
- Due to the same reason, memory usage of our modified version of Byzantine agreement is increased. Although this increase suffers by a factor of 2 in average, we argue that the trade-off is worthwhile. A closer look at Figures 11.1 11.2 uncovers that unlike verification where the time complexity of algorithms is generally not high, our crucial issue in synthesis is time and not space. In other words, in synthesis, *we run out of time before we run out of space*. Thus, given the low memory usage of our case study, it is beneficial to increase memory usage by a factor of 2 to gain speed-up by a factor of 4.

11.4 Case Study 3: Byzantine Agreement with Fail-Stop Faults

A *fail-stop* fault is one that halts a process in response to any internal failure and does so before the effects of that failure become visible [SS83]. In this Section, we introduce

	Space		*Time(s)*			
	reachable states	**memory (MB)**	**deadlock resolution** Eliminate	**deadlock resolution** Recovery	**fault-span generation**	**total**
$\mathcal{BA}^5$	10^5	5.2	1	< 1	< 1	1
$\mathcal{BA}^{10}$	10^8	13	28	3	2	36
$\mathcal{BA}^{15}$	10^{12}	14.5	505	15	5	528
$\mathcal{BA}^{20}$	10^{16}	15.9	4322	35	9	4378
$\mathcal{BA}^{25}$	10^{20}	17.8	23387	76	21	23502

(a)



(b)

Figure 11.3: Experimental results for Byzantine agreement subject to fail-stop faults.

fail-stop faults to the Byzantine agreement problem to make the program more complicated (denoted $\mathcal{BAFS}$). To this end, we first add a Boolean variable u to teach process; if u is true then the process is alive and working, otherwise, the process has been stopped and is not working. Thus, actions of a process, say j, are as follows:

$$\mathcal{BAFS}1_j \;::\; (d.j = \bot) \wedge (f.j = \mathit{false}) \wedge (u.j = \mathit{true}) \quad \longrightarrow \quad d.j \;:=\; d.g;$$
$$\mathcal{BAFS}2_j \;::\; (d.j \neq \bot) \wedge (f.j = \mathit{false}) \wedge (u.j = \mathit{true}) \quad \longrightarrow \quad f.j \;:=\; \mathit{true};$$

In addition to the faults introduced in the example in Section 7.2.3, we introduce the following fault action:

$$\forall p :: (u.p) \quad \longrightarrow \quad u.j \ := \ false;$$

In other words, a process may stop working only if all other processes are alive. Consequently, a Byzantine process can change its decision only if it is alive. Thus, we revise fault action F_1 as follows:

$$F_1 \ :: \ b.j \ \wedge \ u.j \qquad \longrightarrow \qquad d.j, \ \ f.j \ := \ 0|1, \ \ false|true;$$

Likewise, the safety specification and invariant of $\mathcal{BAFS}$ must express the fact that a process may decide or finalize its decision only if it has not stopped due to the occurrence of faults. Thus, $SPEC_{bt_{\mathcal{BA}}}$ and $Inv_{\mathcal{BAFS}}$ are as follows:

$$\begin{array}{lcl} SPEC_{bt_{\mathcal{BAFS}}} & = & (\exists p \in \{j,k,l\} \ :: \ \neg b'.g \ \wedge \ \neg b'.p \ \wedge \\ & & \qquad (d'.p \neq \bot) \ \wedge \ f'.p \ \wedge \ (d'.p \neq d'.g)) \ \vee \\ & & (\exists p,q \in \{j,k,l\} \ :: \ \neg b'.p \ \wedge \ \neg b'.q \ \wedge \\ & & \qquad f'.p \ \wedge \ f'.q \ \wedge \ u'.p \ \wedge \ u'.q \ \wedge \\ & & \qquad (d'.p \neq \bot) \ \wedge \ (d'.q \neq \bot) \ \wedge \ (d'.p \neq d'.q)) \ \vee \\ & & (\exists p \in \{j,k,l\} \ :: \ \neg b.p \ \wedge \ \neg b'.p \ \wedge \ f.p \ \wedge \ \neg u'.p \ \wedge \\ & & \qquad ((d.p \neq d'.p) \ \vee \ (f.p \neq f'.p))), \end{array}$$

and

$$\begin{array}{ll} Inv_{\mathcal{BAFS}} = & (\forall p,q \in \{j,k,l\} \ :: \ u.p \ \vee \ u.q) \ \wedge \\ & [\neg b.g \ \wedge \ (\forall p,q \in \{j,k,l\} \ :: \ (\neg b.p \ \vee \ \neg b.q)) \ \ \wedge \\ & (\forall p \in \{j,k,l\} \ :: \ \neg b.p \Rightarrow (d.p = \bot \vee d.p = d.g)) \ \ \wedge \\ & (\forall p \in \{j,k,l\} \ :: \ (\neg b.p \wedge f.p) \ \Rightarrow \ (d.p \neq \bot)) \\ & \vee \\ & (b.g \ \wedge \ \neg b.j \ \wedge \ \neg\, b.k \ \wedge \ \neg\, b.l \ \wedge \\ & \forall p,q \in \{j,k,l\} \ :: \ ((u.p \ \wedge \ u.q) \ \ \Rightarrow \\ & \qquad\qquad ((d.p \ = \ d.q) \ \wedge \ (d.p \ \neq \ \bot))))]. \end{array}$$

Figure 11.3 shows the result of our experiments on Byzantine agreement program subject to fail-stop faults with 5-25 non-general processes. The growth trend of both time and space as the number of processes increases is similar to the normal Byzantine agreement. However, $\mathcal{BAFS}$ has obviously a larger state space and occurrence of fail-stop faults makes the synthesis problem more complex. In particular, state elimination is significantly harder to solve, as the Procedure Eliminate involves more backtracking steps due to existence of a failed process on reachability paths. As a concrete example, observe that it takes about 6.5 hours to synthesize a solution to $\mathcal{BAFS}^{25}$, while it only takes 11 minutes to synthesize a solution to $\mathcal{BA}^{25}$.

11.5 Case Study 4: Token Ring

In a token ring program for solving distributed mutual exclusion (denoted $\mathcal{TR}$), processes $0..N$ are organized in a ring and the token is circulated along the ring in a fixed direction. Each process, say p where $p \in \{0..N\}$, maintains a variable $x.p$ with domain $\{0, 1, \bot\}$, where $\bot$ denotes a corrupted value. Process p, $0 \leq p \leq N-1$, has the token and can enter the critical section iff $x.p$ differs from its successor $x.(p+1)$ and process N has the token iff $x.N$ is the same as its successor $x.0$. Each process p can only write its local variable (i.e., $x.p$). Moreover, a process can only read its own local variable and the variable of its predecessor. Thus, the read/write restrictions are as follows:

$V_p = \{x.i \mid 0 \leq i \leq N\}$, where $p \in \{0..N\}$,
$W_p = \{x.p\}$, where $p \in \{0..N\}$,
$R_p = \{x.p, x.(p-1)\}$, where $p \in \{1..N\}$, and
$R_0 = \{x.0, x.N\}$.

Fault-intolerant program. The program, consists of two actions. Formally, these actions are as follows:

$$\begin{array}{llll} \mathcal{TR}_p :: & x.p \neq x.(p-1) & \longrightarrow & x.p := x.(p-1); \\ \mathcal{TR}_0 :: & x.0 = x.N & \longrightarrow & x.0 := x.N +_2 1; \end{array}$$

where $p \in \{1..N\}$ and where $+_2$ denotes modulo 2 addition.

Fault action. Faults can restart at most $N-1$ processes. Thus, the fault action for process p, where $p \in \{0..N\}$:

$$F :: \exists i, j \in \{0..N\} \mid (i \neq j) :: \quad (x_i \neq \bot) \wedge (x_j \neq \bot) \quad \longrightarrow \quad x.p := \bot;$$

Safety specification. The safety specification of $\mathcal{TR}$ requires that a process whose state is uncorrupted should not copy the value of a corrupted process. Formally, the safety specification is the following set of bad transitions:

$$SPEC_{bt_{\mathcal{TR}}} = \bigvee_{p=0}^{N} (x.p \neq \bot \ \wedge \ x'.p = \bot).$$

Note that in token ring (unlike Byzantine agreement), we require that the safety specification can only be violated by execution of program actions. In other words, when a fault action restarts a process, safety is not violated.

Invariant predicate. Finally, the invariant predicate of the token ring problem is determined as follows. Consider a state where a process, say p, has the token. In this state, since no other process has the token, the x-value of all processes $0..p$ is identical and the x-value of all processes $(p+1)..N$ is identical. Letting X denote the string of binary values $x.0, x.1 \cdots x.N$, we have that X satisfies the regular expression $(0^l 1^{(N+1-l)} \cup 1^l 0^{(N+1-l)})$, which denotes a sequence of length $N+1$ consisting of zeros followed by ones or ones followed by zeros.

Fault-tolerant program. The output of our algorithm is a program $\mathcal{TR}'$ that tolerates the above fault action. Intuitively, a process in the synthesized program is allowed to copy the value of its predecessor, if this value in not corrupted. The actions of the synthesized fault-tolerant program are as follows:

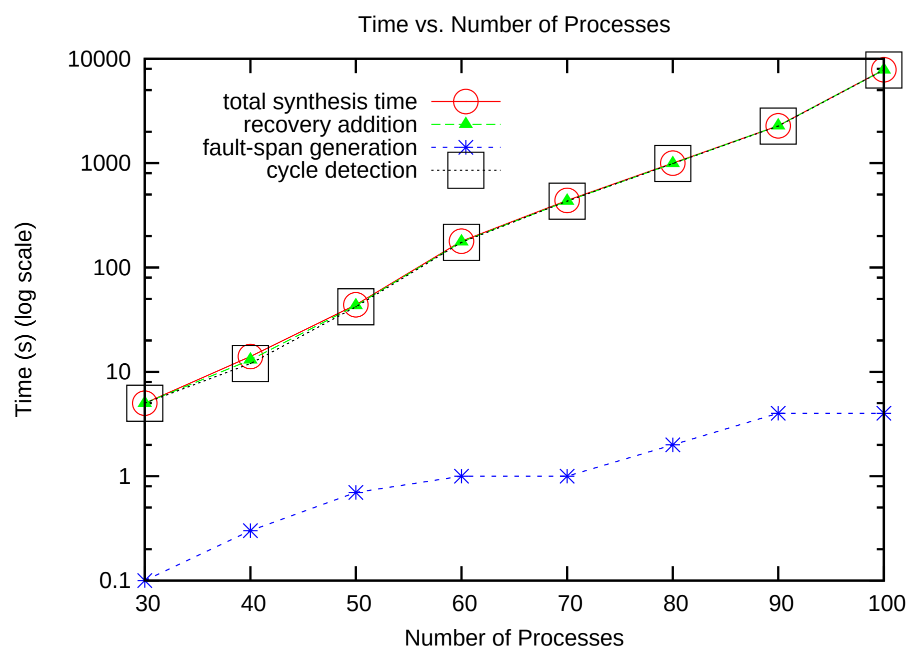
$$\begin{array}{llll} \mathcal{TR}'_p & :: \ (x.p \neq x.(p-1)) \ \wedge \ (x.(p-1) \neq \bot) & \longrightarrow & x.p \ := \ x.(p-1); \\ \mathcal{TR}'_0 & :: \ (x.0 \neq (x.N +_2 1)) \ \wedge \ (x.N \neq \bot) & \longrightarrow & x.0 \ := \ x.N +_2 1; \end{array}$$

where $p \in \{1..N\}$. Observe that action $\mathcal{TR}'_0$ stipulates recovery transitions that start from outside program invariant as well.

Figure 11.4 shows the results of our experiments with respect to the token ring program. Although token ring has a less complex structure than Byzantine agreement, it exhibits features that Byzantine agreement does not. One of these features is the existence of two cycles in both input and output programs which affects the addition of multi-step recovery. Another feature is concerned with the size of fault-span. Unlike Byzantine agreement, the fault-span (i.e., the set of all reachable states) of token ring is almost identical to the state space of the program.

	Space		*Time(s)*			
	reachable states	**memory (MB)**	**recovery addition**	**cycle detection**	**fault-span generation**	**total**
$\mathcal{TR}^{30}$	10^{14}	8	5	5	< 1	5
$\mathcal{TR}^{40}$	10^{19}	13.3	13	13	< 1	35
$\mathcal{TR}^{50}$	10^{23}	14.5	43	43	< 1	44
$\mathcal{TR}^{60}$	10^{28}	15.2	176	174	1	179
$\mathcal{TR}^{70}$	10^{33}	18.8	433	432	1	438
$\mathcal{TR}^{80}$	10^{38}	25.3	992	990	2	999
$\mathcal{TR}^{90}$	10^{42}	33.7	2272	2270	4	2283
$\mathcal{TR}^{100}$	10^{47}	44.2	7824	7819	4	7837

(a)



(b)

Figure 11.4: Experimental results for token ring mutual exclusion program.

Total synthesis time

Similar to the previous case studies, in case of token ring, the total synthesis time is also sublinear to the number of reachable states. We emphasize that the result of our experiments with respect to token ring is considerably different from the results reported in [BK07b]; we can synthesize up to 100 processes in less than two hours while it takes 8 hours to synthesize 25 processes using the method in [BK07b]. Besides, obvious optimizations in the core algorithm, the most effective factor contributing in such a dramatic improvement is the approach in reachability analysis. BDD-based computation of reachable states is normally achieved using a breadth-first search algorithm on state-transition graph of the input program. Such a BFS algorithm involves a frontier generation step which can be implemented in two ways:

1. Applying the transition predicate only on unexplored states which at iteration d, consists of all states at distance exactly d from the invariant predicate.

2. Applying the transition predicate to all known states, that is all states at distance at most d from the invariant predicate.

While the second approach may sound wasteful the cost of applying the transition predicate in a symbolic setting depends on the number of nodes in the corresponding BDD, not on the number of states encoded by the BDD. Thus, even in the verification research community, it is unknown which approach is better. In [BK07b], we implemented the first approach, but through analyzing more case studies, we choose to incorporate the second approach for its generality. An obvious reason for better performance of the second approach in the context of our synthesis problem that it is highly probable that the entire state space of programs is reachable in the presence of faults. Thus, many variables may become don't care in the corresponding BDD which in turn reduces the number of nodes in the BDD. Moreover, in programs such as token ring, there exists many transitions that visits states that are already explored. Consequently, the issue of fault-span generation is not a serious bottleneck as it was in [BK07b].

Deadlock resolution and cycle detection

In this case study, we are not concerned with state elimination time, as safe recovery from all deadlock states is possible. Thus, in order to analyze deadlock resolution time, we only focus on addition of recovery. As can be seen in Figure 11.4, the total synthesis time is almost equal to the time spent for adding recovery paths to the program. Moreover, the time spent for adding recovery is almost equal to the time spent for detecting cycles. First, we note that in previous case studies, cycle detection time were negligible and, hence, was not discussed. However, in this case study, a considerable amount of time is spent for detecting cycles. Recall that in Procedure AddRecovery, after adding a new layer to recovery paths, we check whether or not a cycle has been introduced to the fault-span of the intermediate program (see Line 5). Since the input program in previous case studies does not contain a cycle, the cycle detection algorithm tends to return a negative answer fairly fast. However, one can easily observe that the token ring intolerant program has two cycles that cover all states in the invariant. Thus, in the steps of adding multi-step recovery paths, new cycles are introduced to the faults-span symmetrically which is reflected in the time spent to detect cycles and subsequently removing transitions involved in the cycles. In fact, one can observe in Figure 11.4-b the total synthesis times, and time spent detecting cycles and adding recovery are almost equal. Thus, in programs such as token ring cycle detection becomes the stumbling block of our synthesis algorithm.

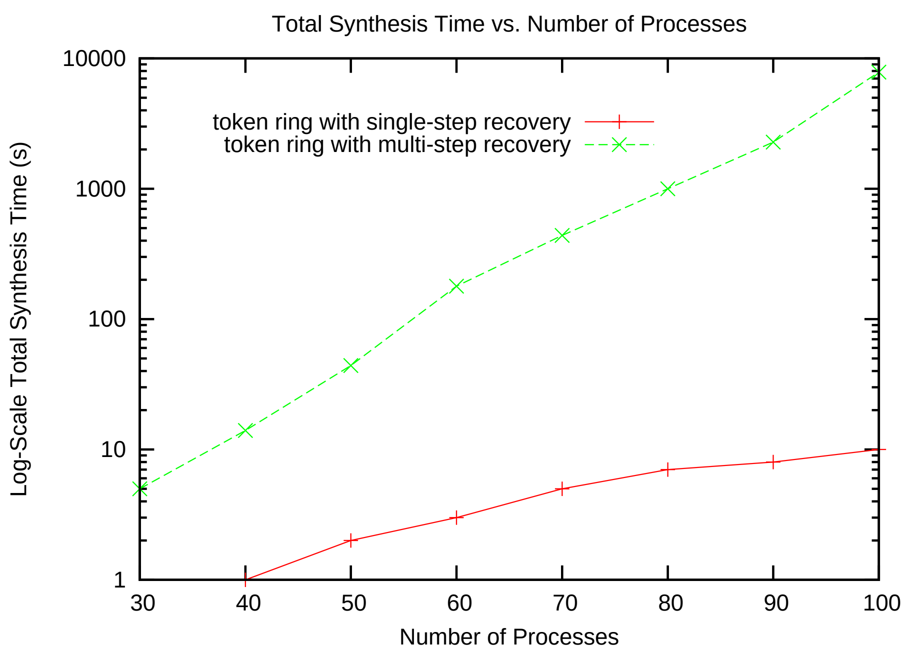
11.5.1 The Effect of Multi-Step Recovery

The issue of cycle detection exists in addition of recovery, as our algorithm constructs *multi-step* recovery paths. Notice that the first recovery step includes transitions that originate from a set of reachable deadlock states and end in the invariant predicate. Recall that each transition has to be added along with its corresponding group predicate. Thus, including additional recovery steps can potentially introduce cycles to the fault-span which in turn prohibits the program to recover to the invariant predicate in a finite number of steps. Hence, an algorithm that synthesizes single-step recovery to an input program need not detect cycles.

Obviously, depending upon the structure of input program, a different type of recovery path may be required. For instance, in case of the token ring program, single-step recovery suffices to resolve all deadlock states. Thus, a respective algorithm need not include the while loop and DetectCycle function in the Procedure AddRecovery. Figure 11.5 shows the result of experiments using such an algorithm for adding single-step recovery to $\mathcal{TR}$. As can be seen, an enormous speed-up is gained. Precisely, in average, the total synthesis time drops by a factor of 330. To illustrate the effect of such a small change in the algorithm, we note that one can synthesize token ring with 200 processes (reachable states of size 10^{95}) in less than 2 minutes.

	Space		*Time(s)*			
	reachable states	**memory (MB)**	**recovery addition**	**cycle detection**	**fault-span generation**	**total**
$\mathcal{TR}^{30}$	10^{14}	4	< 1	0	< 1	0
$\mathcal{TR}^{40}$	10^{19}	5	< 1	0	< 1	1
$\mathcal{TR}^{50}$	10^{23}	7.6	1	0	< 1	1
$\mathcal{TR}^{60}$	10^{28}	9.8	1	0	< 1	2
$\mathcal{TR}^{70}$	10^{33}	11.4	1	0	1	3
$\mathcal{TR}^{80}$	10^{38}	12.5	1	0	3	5
$\mathcal{TR}^{90}$	10^{42}	12.9	1	0	4	6
$\mathcal{TR}^{100}$	10^{47}	13	2	0	4	8

(a)



(b)

Figure 11.5: Experimental results for token ring mutual exclusion with single-step recovery.

11.6 Case Study 5: Infuse

We now focus on Infuse, a time division multiple access (TDMA) based reliable data dissemination protocol in sensor networks [KA06]. Our intention to present this case study is twofold. First, we intend to demonstrate an application of our algorithm outside the literature of fault-tolerant distributed computing. In other words, we demonstrate the applicability of our algorithm in real-world problems by adding fault-tolerance to a sensor network protocol. Secondly, we use Infuse as yet another case study to analyze the performance of our algorithm.

In Infuse, a base station is responsible for communicating with the outside world. The data is split into fixed size packets. Note that Infuse is not concerned with the contents of the data. In our version of case study, all sensors are located in a simple line topology. The base station sends new data to its sole neighbor. Then, this neighbor forwards the packet to its neighbor and so on.

Each sensor maintains two variables r and s where r denotes the sequence number of the last packet the sensor has received and s denotes the sequence number of the packet to be sent to its neighbor. Each variable ranges over $0..M$, where M is the number of bytes in each packet. Thus, if sensors are numbered 0 to N, where sensor 0 is the base station, the read/write restrictions for corresponding processes are as follows:

$R_0 = \{s.0, r.0, s.1, r.1\}$,
$W_0 = \{s.0, r.0\}$,

$R_j = \{s.(j-1), r_(j-1), s.j, r.j, s.(j+1), r.(j+1)\}$,
$W_j = \{s.j, r.j\}$, where $1 \leq j \leq N-1$,

$R_N = \{s.N, r.N, s.(N-1), r.(N-1)\}$,
$W_N = \{s.N, r.N\}$.

Fault-intolerant program. Initially, all packets are disseminated from the base station. A new packet is sent to sensor 1 when base station knows that sensor 1 has received the last packet it had sent. Thus, the action for the base station is as follows:

$$\mathcal{IF}_0 \ :: \ s.0 = r.1 \quad \longrightarrow \quad s.0 \ := s.0 + 1;$$

Intuitively, a sensor, say j, where $j \in \{1..N-1\}$, may receive a new packet, if

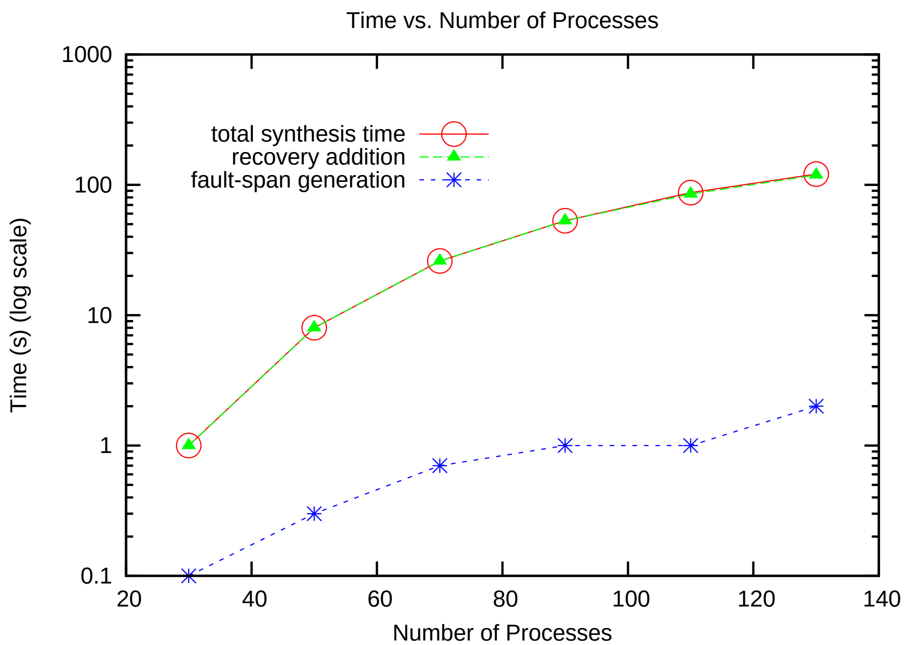
1. the successor (i.e., $j+1$) has all the packets that j has, and

2. $j-1$ is transmitting the next packet.

If this is the case, then j receives the packet and transmits it in the next slot. Thus, the

	Space		*Time(s)*		
	reachable states	**memory (MB)**	**recovery addition**	**fault-span generation**	**total**
$\mathcal{IF}^{30}$	10^{15}	4.7	1	< 1	1
$\mathcal{IF}^{50}$	10^{25}	10	8	< 1	8
$\mathcal{IF}^{70}$	10^{34}	11	26	< 1	26
$\mathcal{IF}^{90}$	10^{43}	14.5	53	1	53
$\mathcal{IF}^{110}$	10^{51}	15	85	1	87
$\mathcal{IF}^{130}$	10^{60}	16	118	2	121

(a)



(b)

Figure 11.6: Experimental results for Infuse bulk data dissemination protocol in sensor networks.

action that models packet transmission for sensors $1..N-1$ is as follows:

$$\mathcal{IF}_j \ :: \ (r.j = r.(j+1)) \ \wedge \ (s.(j-1) = r.j+1) \qquad \longrightarrow \qquad r.j, \ s.j \ := \ r.j+1, \ s.j+1;$$

Finally, sensor N obtains a packet, if its predecessor has the next packet that sensor N expects. Formally,

$$\mathcal{IF}_N \ :: \ s.(N-1) = r.N+1 \ \longrightarrow \ r.N \ := \ r.N+1;$$

Fault actions. A fault causes the base station to transmit a packet that sensor 1 does not expect. In other words, some packet transmitted by the based station is lost. For the rest of sensors, a fault allows a sensor, say j, to receive a packet from its predecessor $j-1$ even though its successor $j+1$ did not obtain the packet that j transmitted last. Thus, the fault actions of Infuse are as follows:

$$
\begin{array}{lll}
F_0 \ : \ \mathit{true} & \longrightarrow & s.0 \ := \ s.0 \ + \ 1; \\
F_j \ :: \ (r.j \leq r.(j-1)) \ \wedge \ (s.(j-1) = r.j+1) & & \\
 & \longrightarrow & r.j, \ s.j \ := \ r.j+1, \ s.j+1;
\end{array}
$$

Safety specification. The safety specification of Infuse informally consists of the following constrains:

- Transmission of a packet cannot be undone and packets can be sent in a subsequence,
- A sensor is not allowed to receive a packet unless its predecessor neighbor has it,
- A sensor may not send a packet that it has not obtained yet.
- Finally, the current s-value of a sensor should reflect the packet it expects from the neighboring sensor.

Thus, the safety specification of Infuse is formally as follows:

$$
\begin{array}{rcl}
SPEC_{bt_{\mathcal{IF}}} & = & \exists p \in \{1..N\} \ :: \ (r'.p \ < \ r.p) \ \vee \ (r'.p \ > \ r.p \ + \ 1) \ \vee \\
 & & \exists p \in \{1..N-1\} \ :: \ (r'.p \ = \ r.p \ + \ 1) \wedge \\
 & & \quad ((r'.p \neq s.(p-1)) \ \wedge \ (r'.p \neq s.(q+1))) \vee \\
 & & (r'.N = r.N+1 \ \ \wedge \ \ r'.N \neq s.(N-1)) \vee \\
 & & \exists p \in \{0..N\} \ :: \ (r'.p \ < \ s'.p) \vee \\
 & & \exists p \in \{0..N-1\} \ :: \ (s.p \ > \ r.(p+1)+1) \wedge \\
 & & \quad (s'.p \ < \ r.(p+1)+1).
\end{array}
$$

Invariant. Informally, the invariant of Infuse specifies the following set of legitimate states:

- It is illegitimate for a sensor to send a pack that it has not received.

- The packet to be sent by a sensor must be expected by its successor sensor.
- The base station initially owns all the packets.
- Finally, a sensor should not have a packet that its predecessor sensor dose not.

Thus, the invariant of Infuse is formally defined as follows: is as follows:

$$
\begin{aligned}
Inv_{\mathcal{IF}} \;=\; & (\forall p \in \{0..N\} \;::\; s.p \;\leq\; r.p) \wedge \\
& (s.0 \;\leq\; r.1 \;+\; 1) \wedge \\
& (\forall p \in \{1..N-1\} \;::\; (s.q \;\leq\; r.(p-1)+1) \wedge \\
& \qquad (s.p \;\leq\; r.(p+1) \;+\; 1)) \wedge \\
& (s.N \;\leq\; r.(N-1) \;+\; 1) \wedge \\
& (r.0 \;=\; M) \wedge \\
& (\forall p \in \{1..N\} \;::\; r.p \;\leq\; r.(p-1))
\end{aligned}
$$

Fault-tolerant program. Given $\mathcal{IF}$, F, $SPEC_{bt_{\mathcal{IF}}}$, and $Inv_{\mathcal{IF}}$, the output of our algorithm is a fault-tolerant version of Infuse, denoted $\mathcal{IF}'$. Adding fault-tolerance to $\mathcal{IF}$ basically results in synthesizing recovery paths. This is due to the fact that occurrence of faults does not lead the program to a state from where safety may be violated. Hence, the only task Algorithm Add_Symbolic_FT needs to accomplish is to guarantee deadlock freedom. And, such deadlock freedom can be achieved by adding safe recovery and no state elimination is required. Formally, the fault-tolerant of Infuse is the following program:

$$
\begin{aligned}
\mathcal{IF}'0_j \;&::\; (r.j = r.(j+1)) \;\wedge\; (s.(j-1) = r.j + 1) \\
&\qquad \longrightarrow \qquad r.j,\; s.j \;:=\; r.j+1,\; s.j+1; \\
\mathcal{IF}'1_j \;&::\; (s.j > r.(j+1)+1) \\
&\qquad \longrightarrow \qquad s.j \;:=\; r.(j+1)+1; \\
\mathcal{IF}'2_j \;&::\; (s.j > r.(j+1)+1) \;\wedge \\
&\quad (s.(j-1) = r.j \;+\; 1) \\
&\qquad \longrightarrow \qquad s.j,\; r.j \;:=\; r.(j+1)+1,\; r.j+1;
\end{aligned}
$$

where $j \in \{1..N-1\}$. Actions of the base station and sensor N can be derived similarly. Observe that $\mathcal{IF}'0_j$ is an unchanged action. Actions $\mathcal{IF}'1_j$ and $\mathcal{IF}'2_j$ are recovery actions and resolve reachable deadlock states. Essentially, these actions enable the program to retransmit packets that are lost while maintaining the safety specification to keep the correct sequence of packet transmission.

Figure 11.6 shows the result of our experiments with respect to Infuse. We note that $\mathcal{IF}$ does not reach states that need to be eliminated. In addition, the state-transition graph of Infuse does not include cycles. Thus, cycle detection and state elimination do not play any role in adding fault-tolerance to $\mathcal{IF}$. Given these facts, Figure 11.6 is self-evident in describing the behavior of Algorithm Symbolic_Add_FT with respect to Infuse. The majority of total synthesis time is spent to add safe recovery which is expected due to the structure of Infuse. One can also observe that the fault-span generation time is negligible. This is due to the fact that the diameter of the state-transition graph of Infuse is short.

Chapter 12

The Tool SYCRAFT

This chapter describes the tool SYCRAFT (*SYmboliC synthesizeR and Adder of Fault-Tolerance*). In SYCRAFT, a distributed fault-intolerant program is specified in terms of a set of processes and an invariant. Each process is specified as a set of actions in a guarded command language, a set of variables that the process can read, and a set of variables that the process can write. Given a set of fault actions and a specification, the tool transforms the input distributed fault-intolerant program into a distributed fault-tolerant program via a symbolic implementation of respective algorithms.

The tool has been tested using various case studies. These case studies include problems from the literature of fault-tolerant computing in distributed systems (e.g., Byzantine agreement, Byzantine agreement with fail-stop faults, Byzantine agreement with constrained specification, token ring, triple modular redundancy, alternating bit protocol) as well as examples from research and industrial institutions (e.g., an altitude switch controller [BH00], and a data dissemination protocol in sensor networks [KA06]). SYCRAFT is written in C++ and the symbolic algorithms are implemented using the Glu/CUDD 2.1 package[1]. The source code of SYCRAFT is freely available and can be downloaded from `http://www.cse.msu.edu/~borzoo/sycraft`. Installation instructions and other resources are also available in the web site. The tool has been tested on Sun Solaris, Mac OS, and various distributions of Linux (e.g., Debian, Ubuntu, and Fedora).

This chapter is organized as follows. First, in Section 12.1, we present the grammar for input programs to SYCRAFT. Then, in Sections 12.4-12.6, we present three examples for

[1]Details about Colorado University Decision Diagram Package are available at `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`.

specifying programs, invariants and safety specifications in SYCRAFT.

12.1 SYCRAFT Input Program Language and Grammar

Every input program to SYCRAFT has eight declaration sections. These sections are meant to declare program name, constants, variables, processes, faults, invariant, safety specification, and finally a transition predicate which the output program is prohibited to execute.

```
<prog> ::=     <progdecl> <constdecl> <vardecl> <procdecl>
               <faultdecl> <invariantdecl> <specdecl> <prohibited>
```

```
<progdecl> ::= "program" <identifier> ";"
```

The rule <identifier> is reduced by SYCRAFT's lexical analyzer and returned as a token. An identifier can be any combination of alphanumeric characters. The only constraints are an identifier has to start with an alphabet and case matters. A typical structure of a SYCRAFT input program is illustrated bellow:

```
program MyProgram;

const
...
var
...
process p1
...
endprocess
process p2
...
endprocess
fault MyFaults
...
endfault
invariant
...
specification
...
prohibited
...
```

12.1.1 Program, Constant, and Variable Declaration

We now describe each declaration section by starting from constants. *Constants* are often used to facilitate parameterizing Booleans and integers such as domain of variables, number of processes, or range of quantifiers.

```
<constdecl> ::=    "const" <constlist> | <empty>
<constlist> ::=    <concreteconst> | <constlist> <concreteconst>
```

```
<concreteconst>::=  "int" <identifuer> "=" <guard> ";" |
                    "boolean" <identifier> "=" <guard> ";"
<empty> ::=
```

We explain the rule <guard> in detail in the next section. For now, a guard can be any combination of arithmetic and Boolean expressions, including a single integer or Boolean. For example, the following is a constant declaration:

```
const
        int N = 2;
        int bot = 2;
        int M = N - 1;
```

In SYCRAFT, *variables* are of two types: nonnegative integer and Boolean. Note that SYCRAFT does not have a perfect type checker and it is user's responsibility to ensure correct type checking. Each integer must be declared along with its domain. The lower-bound of an integer domain must be zero, but its upper bound is optional. Variables can be defined as an array. The index of the first element of an array is always zero.

```
<vardecl> ::=       "var" <varlist>
<varlist> ::=       <concretevar> | <varlist> <concretevar>
<concretevar> ::=   <booldecl> | <intdecl>
<booldecl> ::=      "boolean" <identifuer> ";" |
                    "boolean" <identifier> "." <range> ";"
<range> ::=         <rangedelim> ".." <rangedelim>
<rangedelim> ::=    <guard>
<intdecl> ::=       "int" <indetifier> ": domain" <range> ";" |
                    "int" <identifier> "."<range>": domain" <range> ";"
```

For example, the following is a variable declaration.

```
var
        boolean bg;
        boolean b.0..N;
        boolean f.0..N;
        int dg: domain 0..1;
        int d.0..N: domain 0..2;
```

One can refer to elements of array d declared above, by d.0, d.1, ..., d.N. One can also use expressions such as d.(i+1) where i is an integer ranges over 0..N-1. We explain parameterized variable elements in the next section.

12.1.2 Process Declaration and Structure

As mentioned earlier, a distributed program consists of a set of *processes*. In SYCRAFT, each process includes:

1. A set of *process actions* given in *guarded commands*,
2. A *fault section* which accommodates fault actions that the process is subject to,
3. a *prohibited* section which defines a set of transitions that the process is not allowed to execute,
4. a set of variables that the process is allowed to *read*, and
5. a set of variables that the process is allowed to *write*.

The syntax of actions is of the form $g \rightarrow st$, where the guard g is a Boolean expression and statement st is a set of (possibly non-deterministic) assignments. The semantics of actions is such that at each time, one of the actions whose guard is evaluated as *true* is chosen to be executed non-deterministically. The read-write restrictions model the issue of distribution in the input program. Note that in SYCRAFT, fault actions are able to read and write all the program variables. Prohibited transitions are specified as a conjunction of an (unprimed) source predicate and a (primed) target predicate.

<procdecl> ::= <procstruct> | <procdecl> <procstruct>

<procstruct> ::= "process" <identifier> <procrange> <faultdecl>
<prohibited> <rwrestrict> "endprocess"

<procrange> ::= ":" <range> | <empty>

For example, a process declarations template without range is the following:

```
process MyProcess

        guarded commands
        ...
        fault actions
        ...
        prohibited transitions
        ...
        read/write restrictions

endprocess
```

Likewise, an array of symmetric processes can be defined as follows:

```
process j:0..N
...
endprocess
```

We emphasize that fault actions and prohibited transitions sections are allowed to be empty in a process. The reason for providing the feature of defining an array of processes is due to the fact that many distributed programs consist of identical processes that can be modeled through parametrization. We now describe constituents of a process. First, the set of guarded commands which essentially model the behavior of a process.

<actions> ::= <actionlist>

<actionlist> ::= <action> | <actionlist> “[]” <action>

<action> ::= <guard> “- ->” <statement> “;”

The following is an example of two guarded commands (actions) joined by “[]” operator:

```
(d.j == bot) & !f.j & !b.j --> d.j := dg;
[]
(d.j != bot) & !f.j & !b.j --> f.j := true;
```

We now describe each nonterminal of the above rules for defining a process. The first rule is <guard>. A guard is essentially a combination of quantified Boolean and arithmetic expressions.

<guard>::= <guard> “&” <guard> |
<guard> “|” <guard> |
<guard> “=>” <guard> |
<guard> “ ^ ” <guard> |
“!”<guard> |
“exists” <boundlist> “in” <range>
<quantcond> “:: (” <guard> “)” |
“forall” <boundlist> “in” <range>
<quantcond> “:: (” <guard> “)” |
<arithmaticexp> |
<comparison>

<boundlist> ::= <identifier> | <boundlist> "," <identifier>

<quantcond> ::= ": (" <guard> ")" | <empty>

The first five rules of "guard" are concerned with logical and, or, implication, xor, and negation operators, respectively. The next two rules are concerned with universal and existential quantifiers, respectively. A quantifier has a list of bound variables separated by comma. All variables must be from the same range. Moreover, a quantifier may have two conditions determined by the internal nonterminals <quantcond> and <guard>. The first condition reduced by rule <quantcond> is on bound variables. The second condition is on which SYCRAFT's compiler eliminates the quantifier. Although the above grammar allows nested quantifiers, we emphasize that SYCRAFT's code generator currently does not support nested quantifiers. In other words, an input program containing nested quantifiers may compile successfully, but the corresponding BDDs would encode a different expression. We describe the syntax of arithmetic and comparison expressions later in this section. The following is an example of a Boolean expression without arithmetic expression:

```
!bg  &  (forall p, q in 0..N:  (p != q)  ::  (!b.p  |  !b.q))
```

where is N is a constant. We note that in SYCRAFT, equality of Boolean variables has to be expressed using Boolean operators only. For instance, the following expressions are not meaningful to SYCRAFT:

- (x == **true**),
- (x == **false**),
- (x == y), and
- (x != y).

The right way to write the above expressions in SYCRAFT is respectively as follows:

- x,
- !x,

- !(x^y), and
- (x^y).

In addition to logical operators, expressions can contain arithmetic expressions and integer comparisons as well. Arithmetic expressions may include addition, subtraction, and modulo operators. Comparisons may involve equality, inequality, greater than, less than, greater than or equal, and less than or equal comparisons. As mentioned earlier, equality and inequality are not meant to be used for Boolean variables.

```
<arithmaticexp> ::=   <arithmaticexp> “+” <arithmaticexp> |
                      <arithmaticexp> “-”  <arithmaticexp> |
                      <arithmaticexp> “%” <arithmaticexp> |
                      “(” <guard> “)” |
                      <number> |
                      <id> |
                      “true” |
                      “false”

<comparison> ::=      <arithmaticexp> “==” <arithmaticexp> |
                      <arithmaticexp> “!=”  <arithmaticexp> |
                      <arithmaticexp> “>”   <arithmaticexp> |
                      <arithmaticexp> “<”   <arithmaticexp> |
                      <arithmaticexp> “>=” <arithmaticexp> |
                      <arithmaticexp> “<=” <arithmaticexp> |

<id> ::=              <identifier> |
                      <identifier> “ ‘ ” |
                      <identifier> “.”  <arithmaticexp> |
                      <identifier> “.”  <arithmaticexp> “ ‘ ”
```

The following is an example of an expression that involves both Boolean and arithmetic operators along with comparisons:

```
forall q in 1..K :: ((s.q <= r.(q-1) + 1) & (s.q <= r.(q+1) + 1))
```

where s and r and two arrays of integers and K is a constant.

In an expression a (Boolean or integer) variable can appear in four different forms (cf. the <id> rule). A variable can be either primed or unprimed. A primed variable is meant to express the *next-state* value of the variable. For example, if x is a Boolean variable, (x & !x') expresses a transition where x is true in the source state, but it becomes false in the target state. A variable may also be indexed or unindexed. For instance, bg is unindexed. An indexed variable is one that is declared in an array and is used along with an arithmetic expression as parameter. For instance, d.(i+1) is an indexed variable which refers to element (i+1) in array d where i can be a process range, quantifier, or simply a constant. We note that if an indexed variable is primed, its index must appear in parentheses and the prime character must be place after the right parenthesis. For instance, b.i' is a meaningless primed index variable. The right way to write this variable is b.(i)'. Another example of a primed indexed variable is d.(i-1)'.

Finally, a guarded command ends with a statement, which is a set of deterministic or non-deterministic assignments.

```
<statement> ::=  "(" <statement> ")" |
                 <statement> "[]" <statement> |
                 <statement> "," <statement> |
                 <statementstruct>

<statementstruct> ::= <id> ":=" <arithmaticexp>
```

Precisely, given an action, a non-deterministic assignment (i.e., , operator) stipulates that one assignment is chosen non-deterministically when the guard of the action holds. For example, a guarded command with non-deterministic assignment is as follows:

```
(b.j)   -->   (d.j := 1) [] (d.j := 0) [] (f.j := false) [] (f.j := true);
```

To the contrary, in a deterministic assignment, all assignments are considered to determine the next state. The following guarded command contains deterministic assignments.

```
(cs != 2)  &  (br == 0)   -->   (cs := 2), (rr := true);
```

We note that although the SYCRAFT grammar allows a combination of deterministic and non-deterministic assignments, such combinations are not handled during translation of guarded commands to BDDs.

A fault section inside a process expresses the set of fault transitions that the process is subject to. Such fault transitions are modeled using guarded commands. Thus, fault and process actions have exactly the same syntax.

<faultdecl> ::= "faults" <identifier> <actions> "endfaults" | <empty>

Recall that in addition to each process of a program, the program itself may have a fault section as well. These section are allowed to be empty, but there must exist at least one nonempty fault section. Otherwise, it means that the program is subject to no faults and, hence, adding fault-tolerance becomes irrelevant.

Finally, we declare the read/restrictions of a process.

<rwrestrict> ::= "read:" <rwlist> ";"
"write:" <rwlist> ";"

<rwlist> ::= <idrange> | <rwlist> "," <idrange>
<idrange> ::= <id> | <identifier> "." <range>

For example, the following is a legal declaration of read/write restrictions:

```
read: d.0..N, dg, f.j, b.j;
write: d.j, f.j;
```

12.1.3 Invariant, Safety Specification, and Prohibited Transitions Declaration

Invariant predicate, safety specification, and prohibited transitions are simply a combination of Boolean and arithmetic expressions. The invariant predicate has a triple role: it (1) is a set of states from where execution of the program satisfies its safety specification (described below) in the absence of faults, (2) must be closed in the execution of the input program and, (3) specifies the *reachability* condition of the program, i.e., if occurrence of faults results in reaching a state outside the invariant, the (synthesized) fault-tolerant program must *safely* reach the invariant predicate in a finite number of steps. In order to increase the chances of successful synthesis, it is desired that SYCRAFT is given the weakest possible invariant.

Our notion of specification is based on the one defined by Alpern and Schneider [AS85]. The specification section describes the *safety* specification as a set of *bad prefixes* that should not be executed neither by a process nor a fault action. Currently, the size of such prefixes in SYCRAFT is two (i.e., a set of transitions). Since safety specification and prohibited transitions model a set of bad transitions that should not occur in any program computation, they typically involve primed variables to model states that should not be reached. The input program and its processes may or may not have prohibited transitions.

`<invariantdecl> ::=` "invariant" `<guard>` ";"

`<specdecl> ::=` "specification" `<guard>` ";" | `<empty>`

`<prohibited> ::=` "prohibited" `<guard>` ";" | `<empty>`

We emphasize that the main difference between transitions specified in prohibited and specification sections is that transitions in prohibited section should not be executed by the program only. In other words, it is perfectly legitimate for a fault action to executes a transition that is specified in the prohibited section. To the contrary, transitions specified in the specification are not allowed to be executed by both program and fault actions. We typically use prohibited transitions to specify transition that cannot be used in order to add recovery paths.

12.1.4 Operator Precedence

The precedence of operators and their associativity in SYCRAFT are as follows:

:=	left
ˆ	left
=>	left
$\vert$	left
&	left
==, !=	left
>, <, <=, >=	left
+, -	left
forall, exists	left
%	left
,	left
[]	left
!	left
.	non-associative
‘	non-associative

12.2 Internal Functionality

SYCRAFT implements three nested symbolic fixedpoint computations presented in Algorithm 11.1 in Chapter 11. The inner fixedpoint deals with computing the set of states reachable by the input intolerant program and fault transitions. The second fixedpoint computation identifies the set *ms* of states from where the safety condition may be violated by fault transitions. It makes *ms* unreachable by removing program transitions that end in a state in *ms*. Note that in a distributed program, since processes cannot read and write all variables, each transition is associated with a *group* of transitions. Thus, removal or addition of a transition must be done along with its corresponding group. The outer fixedpoint computation deals with resolving *deadlock* states by either (if possible) adding

```
$ bin/sycraft examples/ByzantineAgreement.fin

Initializing MDD manager Glu2.1...
Compiling the input fault-intolerant program...
Creating the symbol table...
Creating intolerant program MDDs...
Input program compiled successfully

Computing ms....
Computing mt....
Running the synthesis algorithms for 3 processes

SAFETY........................OK.
DEADLOCKS.....................OK.
INVARIANT NONEMPTINESS.........OK.
INVARIANT CLOSURE.............OK.
F-SPAN CLOSURE.............FAILED.

Computing the fault-span...
Removing unsafe transitions...
Unsafe transitions removed...
Computing the fault-span...
Step 3-4 Fixpoint reached ...

SAFETY........................OK.
DEADLOCKS.................FAILED.
INVARIANT NONEMPTINESS.........OK.
INVARIANT CLOSURE.............OK.
F-SPAN CLOSURE................OK.

Resolving deadlocks by adding recovery paths
Removing unsafe transitions...
Resolving deadlocks by adding recovery paths
Removing unsafe transitions...
Cycle(s) detected/removed from the fault-span...

SOME RECOVERY ACTION ADDED

Eliminating remaining deadlock states...
Eliminating remaining deadlock states...
Eliminating remaining deadlock states...
Eliminating remaining deadlock states...
Eliminating remaining deadlock states...
Eliminating remaining deadlock states...
Eliminating remaining deadlock states...
Something was eliminated.

SAFETY........................OK.
DEADLOCKS.................FAILED.
INVARIANT NONEMPTINESS.........OK.
INVARIANT CLOSURE.............OK.
F-SPAN CLOSURE................OK.

Computing the fault-span...
Removing unsafe transitions...
Step 3-4 Fixpoint reached ...

SAFETY........................OK.
DEADLOCKS.................FAILED.
INVARIANT NONEMPTINESS.........OK.
INVARIANT CLOSURE.............OK.
F-SPAN CLOSURE.............FAILED.

Resolving deadlocks by adding recovery paths
Removing unsafe transitions...
Resolving deadlocks by adding recovery paths
Removing unsafe transitions...
SOME RECOVERY ACTION ADDED

Eliminating remaining deadlock states...
No program transitions removed.

SAFETY........................OK.
DEADLOCKS.....................OK.
INVARIANT NONEMPTINESS.........OK.
INVARIANT CLOSURE.............OK.
F-SPAN CLOSURE.............FAILED.

Computing the fault-span...
Removing unsafe transitions...
Step 3-4 Fixpoint reached ...

SAFETY.....................FAILED.
DEADLOCKS.................FAILED.
INVARIANT NONEMPTINESS.........OK.
INVARIANT CLOSURE.............OK.
F-SPAN CLOSURE.............FAILED.

Resolving deadlocks by adding recovery paths
Removing unsafe transitions...
Resolving deadlocks by adding recovery paths
Removing unsafe transitions...
SOME RECOVERY ACTION ADDED

Eliminating remaining deadlock states...
No program transitions removed.

SAFETY.....................FAILED.
DEADLOCKS.....................OK.
INVARIANT NONEMPTINESS.........OK.
INVARIANT CLOSURE.............OK.
F-SPAN CLOSURE.............FAILED.

Computing the fault-span...
Removing unsafe transitions...
Step 3-4 Fixpoint reached ...
Step 3-5 Fixpoint reached ...
Step 3-7 Fixpoint reached ...
SAFETY........................OK.
DEADLOCKS.....................OK.
INVARIANT NONEMPTINESS.........OK.
INVARIANT CLOSURE.............OK.
F-SPAN CLOSURE................OK.

Total synthesis time = 0.125s

Select the process you wish to see its
fault-tolerant version:
(1)  j:0..2      (example: "j 1")
(2) quit

Output process:

Destroying MDD manager Glu2.1...
$
```

Figure 12.1: A sample run snapshot of SYCRAFT.

safe recovery simple paths from deadlock states to the program's invariant predicate, or, making them unreachable via adding minimal restrictions on the program. We note that the BDD-based implementation of the aforementioned fixedpoint computations does not apply dynamic variable reordering in the current version of SYCRAFT.

Figure 12.1 demonstrates (1) steps of heuristics implemented in SYCRAFT, (2) verification of corresponding concerns such as satisfaction of safety, closure of invariant and *fault-span* (i.e., the set of states reachable by program and fault actions), and nonexistence of deadlock states for each step (denoted by *OK* and *FAILED*), and (3) satisfaction of fixedpoint computations. In particular, the tool has a solution to the synthesis problem when the answer to verification of all above concerns is affirmative.

The problem of synthesizing distributed fault-tolerant programs is known to be NP-complete and SYCRAFT implements a set of heuristics (cf. Algorithm 11.1) to deal with this complexity. Obviously, the heuristics are incomplete in the sense that they may be unable to synthesize a solution while there exists one. Although we have not observed this incompleteness in our case studies, the incompleteness of heuristics may be observed where recovery through a state outside the fault-span is possible.

The performance of the symbolic algorithms that SYCRAFT implements is discussed in detail in Chapter 11 using case studies including the Byzantine agreement and token ring problems with various number of processes. In particular, by applying more advanced techniques than those introduced in Chapter 11 (e.g., exploiting symmetry in distributed processes and state space partitioning) we have been able to synthesize Byzantine agreement with 3 processes in 0.07s and up to 40 processes (reachable states of size 10^{50}) in less than 100 minutes using a regular personal computer.

12.3 Output Format

The output of SYCRAFT is a fault-tolerant program in terms of its actions. SYCRAFT organizes these actions in three categories. *Unchanged actions* entirely exist in the input program. *Revised actions* exist in the input program, but their guard or statement have been strengthened. SYCRAFT adds *Recovery actions* to enable the program to safely converge to its invariant predicate. Notice that the strengthened actions prohibit the program to reach a state from where validity or agreement is violated in the presence of faults. It also prohibits the program to reach deadlock states from where safe recovery is not possible.

12.4 Example 1: Never 7

This example is adapted from M.Sc. thesis of Bastian Braun at University of Mannheim, Germany. Note that in Braun's thesis the objective is to synthesize a failsafe program, where safe recovery to the program invariant is not required, whereas SYCRAFT synthesizes masking fault-tolerant programs where safe recovery is provided in order to ensure that the synthesized program does not deadlock in the presence of faults. Thus, the our program is slightly different from that developed by Braun.

The program's state-transition graph is illustrated in Figure 12.2. This program has 8 states and the safety specification requires that state 7 should never be reached. The invariant predicate of the program is the set $\{0, 1, 2\}$. Program and fault transitions are clear in the figure. It is easy to observe that the program does not reach state 7 in the absence of faults, but it may reach state 7 due to the occurrence of faults. We use SYCRAFT to synthesize a fault-tolerant program, i.e., a program that never reaches state 7 in both absence and presence of faults. For the sake of illustration, we require that potential transitions $(6, 1), (6, 2)$, and $(3, 0)$ should not be used for adding recovery computations.

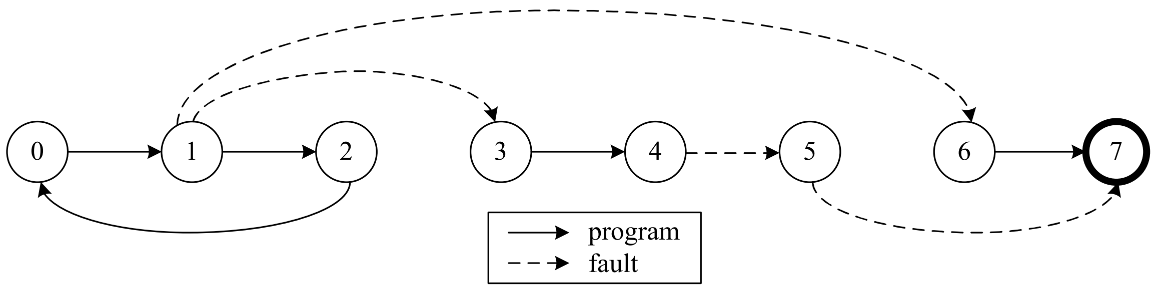


Figure 12.2: Never7 program state-transition graph.

We now model this program in SYCRAFT's input language. We use a variable `state` with domain 0..7 which encodes each state shown in Figure 12.2. We declare and define only one process and this process can read and write the variable `state`, as there is no distribution involved. It is straightforward to see that the guarded commands in `process trans` of Figure 12.3 model the transitions in Figure 12.2. Obviously, `process trans` can read and write variable `state` which concludes the definition of `process trans`.

As mentioned earlier, the invariant predicate of `Never7` consists of state in the set $\{0, 1, 2\}$. It is easy to see that the invariant in Figure 12.3 defines the equivalent expression. The safety specification of our program consists of the simple expression (`state'` $==$ `7`), which essentially includes any transition that ends in state where the value of variable `state` is 7.

Finally, the expression in the `prohibited` section of Figure 12.3, specifies the transitions that we ruled out for adding recovery, i.e., transitions $(6, 1), (6, 2)$, and $(3, 0)$.

After running SYCRAFT on the input file in Figure 12.3, the output is a fault-tolerant version of `Never7` shown in Figure 12.5. As can be seen in this figure, SYCRAFT has removed transitions $(3, 4)$ and $(6, 7)$. Notice that existence of transition $(3, 4)$ may lead the program to state 5 from where occurrence of faults alone violate the safety. Moreover, if the program reaches state 6 due to the occurrence of faults, then transition $(6, 7)$ violates the safety.

```
program never7;

var int state: domain 0..7;
{-----------------------------------------}
process trans
      (state == 0)    -->    state := 1;
     []
      (state == 1)    -->    state := 2;
     []
      (state == 2)    -->    state := 0;
     []
      (state == 3)    -->    state := 4;
     []
      (state == 6)    -->    state := 7;

     fault Malfunction
            (state == 1)    -->    state := 3;
      []
            (state == 1)    -->    state := 6;
      []
            (state == 4)    -->    state := 5;
      []
            (state == 5)    -->    state := 7;
     endfault

     read: state;
     write: state;
endprocess
{-----------------------------------------}
invariant
      (state == 0)  |  (state == 1)  |  (state == 2);
{-----------------------------------------}
specification
      (state'== 7);
{-----------------------------------------}
prohibited
      ((state == 6)  &  ((state' == 1)  |  (state' == 2)))
      |
      ((state == 3)  &  (state' == 0));
```

Figure 12.3: Never7 program as input to SYCRAFT.

One can also notice that SYCRAFT has added three recovery transitions in order to resolve deadlock states. Notice that these transitions do not intersect with transitions specified in the prohibited section. For the reader's convenience, Figure 12.4 shows the state-transition graph of the fault-tolerant version of Never7.

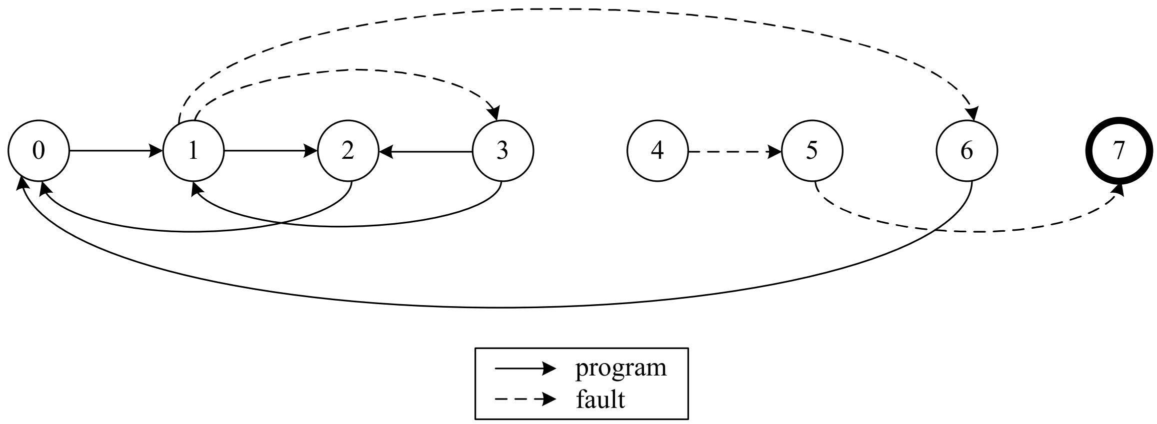


Figure 12.4: Fault-tolerant Never7 state-transition graph.

```
----------------------------------------------
UNCHANGED ACTIONS:
----------------------------------------------
1- (state  ==  0)   -->   (state  :=  1)
2- (state  ==  1)   -->   (state  :=  2)
3- (state  ==  2)   -->   (state  :=  0)
----------------------------------------------
REVISED ACTIONS:
----------------------------------------------
----------------------------------------------
NEW RECOVERY ACTIONS:
----------------------------------------------
1- (state  ==  6)   -->   (state  :=  0)
2- (state  ==  3)   -->   (state  :=  2)
3- (state  ==  3)   -->   (state  :=  1)
----------------------------------------------
```

Figure 12.5: Sycraft output: fault-tolerant Never7.

12.5 Example 2: Token Ring

We now reiterate the description of the token ring problem (cf. Section 11.5) in order to describe the input to Sycraft more clearly. In a token ring program, the processes 0..N are organized in a ring and the token is circulated along the ring in a fixed direction. Each process, say j, maintains a variable x with the domain $\{0, 1, 2\}$, where the value 2 denotes a corrupted value. Process j, where j is greater than zero, has the token iff x.j differs from its successor x.(j + 1) and process N has the token iff x.N is the same as its successor x.0. Each process can only write its local variable (i.e., x.j). Moreover, a process can only read its own local variable and the variable of its predecessor. In this program, faults can restart at most N-1 processes.

The invariant predicate of the token ring problem is determined as follows. Consider a state where a process j has the token. In this state, since no other process has a token, the value of variable x of all processes 0..j is identical and the x value of all processes (j+1)..N is identical. Letting X denote the string of binary values x.0, x.1,..., x.N, we have that X satisfies the regular expression $(0^l.1^{(N+1-l)} \;\cup\; 1^l.0^{(N+1-l)})$, which denotes a sequence of

```
program BinaryTokenRing;
const N = 3;
var int x.0..N: domain 0..2;
{-------------------------------------------}
process p
      (x.0 == 0)  &  (x.N == 0)     -->     x.0 := 1;
     []
      (x.0 == 1)  &  (x.N == 1)     -->     x.0 := 0;

     read: x.0, x.N;
     write: x.0;
endprocess
{-------------------------------------------}
process q: 1..N
      x.q  !=  x.(q - 1)             -->     x.q := x.(q - 1);

     prohibited (x.0 == 2)  &  (x.N == x.(0)')

     read: x.q, x.(q - 1);
     write: x.q;
endprocess
{-------------------------------------------}
fault TokenCorruption
     exists i, j in 0..N : (i != j) :: ((x.i != 2)  &  (x.j != 2))
          -->   (x.0 := 2) [] (x.1 := 2) [] (x.2 := 2) [] (x.3 := 2);
endfault
{-------------------------------------------}
invariant
     ((x.0 == 1) & (x.1 == 0) & (x.2 == 0) & (x.3 == 0)) |
     ((x.0 == 1) & (x.1 == 1) & (x.2 == 0) & (x.3 == 0)) |
     ((x.0 == 1) & (x.1 == 1) & (x.2 == 1) & (x.3 == 0)) |
     ((x.0 == 1) & (x.1 == 1) & (x.2 == 1) & (x.3 == 1)) |
     ((x.0 == 0) & (x.1 == 0) & (x.2 == 0) & (x.3 == 0)) |
     ((x.0 == 0) & (x.1 == 0) & (x.2 == 0) & (x.3 == 1)) |
     ((x.0 == 0) & (x.1 == 0) & (x.2 == 1) & (x.3 == 1)) |
     ((x.0 == 0) & (x.1 == 1) & (x.2 == 1) & (x.3 == 1)) ;
{-------------------------------------------}
prohibited
     exists i in 0..N :: ((x.i != 2)  &  (x.(i)' == 2));
```

Figure 12.6: Token ring program as input to SYCRAFT.

length $N + 1$ consisting of zeros followed by ones or ones followed by zeros.

In token ring processes whose state is uncorrupted are not allowed to copy the value of a corrupted process. Note that in token ring, since this constraint has to be met only by execution of program actions (not fault actions), we specify it under the prohibited section in SYCRAFT.

Figure 12.6 shows how we model the token ring problem in SYCRAFT. Specifically, we define two processes, p and q. Process p models process 0 in the ring and process q which ranges over 1..N models the rest of processes in the ring. Currently, SYCRAFT's grammar does not feature regular expressions. Thus, we have to model them explicitly in SYCRAFT.

An alert reader observes that the above program works correctly in the absence of faults. However, in the presence of faults, it deadlocks due to existence of several tokens or no tokens in the ring. The output of SYCRAFT after adding fault-tolerance to BinaryTokenRing is as follows (cf. Figure 12.7). Intuitively, a q-process in the synthesized program is allowed to copy the value of its predecessor, if this value in not corrupted. Notice that recovery action process p in the synthesized program stipulate recovery mechanism that starts from outside program invariant and reaches the invariant in one step.

12.6 Example 3: Byzantine Agreement

We now reiterate the description of the Byzantine agreement problem (cf. Subsection 2.1.3) in order to describe the input to SYCRAFT more clearly. In Byzantine agreement [LSP82], the program consists of a *general* g and three (or more) *non-general* processes: 0, 1, and 2 Figure 12.8. Since, the non-general processes have the same structure, we model them as a process j that ranges over 0..2. The general is not modeled as a process, but by two global variables bg and dg. Each process maintains a decision d; for the general, the decision can be either 0 or 1, and for the non-general processes, the decision can be 0, 1 or 2, where the value 2 denotes that the corresponding process has not yet received the value from the general. Each non-general process also maintains a Boolean variable f that denotes whether that process has finalized its decision. To represent a Byzantine process, we introduce a variable b for each process; if $b.j$ is true then process j is Byzantine, where process j cannot read $b.k$ for $k \neq j$.

For process q = 1

```
--------------------------------------------------
UNCHANGED ACTIONS:
--------------------------------------------------
--------------------------------------------------
REVISED ACTIONS:
--------------------------------------------------
1- (x0 == 1)  &  (x1 == 2)   -->   (x1 := 1)
2- (x0 == 0)  &  (x1 == 2)   -->   (x1 := 0)
3- (x0 == 1)  &  (x1 == 0)   -->   (x1 := 1)
4- (x0 == 0)  &  (x1 == 1)   -->   (x1 := 0)
--------------------------------------------------
NEW RECOVERY ACTIONS:
--------------------------------------------------
--------------------------------------------------
```

For process p:

```
-------------------------------------------------
UNCHANGED ACTIONS:
-------------------------------------------------
1- ((x0 == 0)  &  (x3 == 0)) -->   (x0 := 1)
2- ((x0 == 1)  &  (x3 == 1)) -->   (x0 := 0)
-------------------------------------------------
REVISED ACTIONS:
-------------------------------------------------
-------------------------------------------------
NEW RECOVERY ACTIONS:
-------------------------------------------------
1- (x0 == 2)  &  (x3 == 0)   -->   (x0 := 1)
2- (x0 == 2)  &  (x3 == 1)   -->   (x0 := 0)
-------------------------------------------------
```

Figure 12.7: SYCRAFT output: fault-tolerant token ring.

The program works as follows. Each non-general process copies the decision value from the general (Line 12) and then finalizes that value (Line 14). A fault action may turn a process to a Byzantine process, if no other process is Byzantine (Line 16). Faults also change the decision (i.e., variables d and f) of a Byzantine process arbitrarily and non-deterministically (Line 18). As mentioned earlier, in SYCRAFT, one can specify faults that are not associated with a particular process. This feature is useful for cases where faults affect global variables, e.g., the decision of the general (Lines 24-28). In the prohibited

```
1)  program Byzantine_Agreement;
2)  const
3)     int N = 2;
4)     int bot = 2;
5)  var
6)     boolean bg;
7)     boolean b.0..N;
8)     boolean f.0..N;
9)     int dg: domain 0..1;
10)    int d.0..N: domain 0..2;
    {- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -}
11) process j: 0..N
12)         (d.j == bot)  &  !f.j & !b.j     - ->    d.j  :=  dg;
13)         []
14)         (d.j != bot)  &  !f.j & !b.j      - ->    f.j  :=  true;
15)         fault Byzantine_NonGeneral
16)            !bg  &  (forall p in 0..N:: (!b.p))  - ->  b.j  :=  true;
17)         []
18)            b.j  - ->  (d.j := 1)  []  (d.j := 0)  []  (f.j := false)  []  (f.j := true);
19)         endfault
20)         prohibited
21)            !b.j  &  !b.(j)'  &  f.j  &  ((d.j != d.(j)')  |  !f.(j)');
22)         read: d.0..N, dg, f.j, b.j;
23)         write: d.j, f.j;
24) endprocess
    {- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -}
25) fault Byzantine_General
26)         !bg  &  (forall p in 0..N:: (!b.p))   - ->  bg  :=  true;
27)         []
28)         bg  - ->  (dg := 1)  []  (dg := 0);
29) endfault
    {- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -}
30) invariant
31)         (!bg  &  (forall p, q in 0..N:(p != q):: (!b.p  |  !b.q))  &
32)         (forall r in 0..N:: (!b.r  =>  ((d.r == bot)  |  (d.r == dg)))) &
33)         (forall s in 0..N:: ((!b.s & f.s)  =>  (d.s != bot))))
34)         |
35)         (bg  &  (forall t in 0..N::
                  (!b.t))  &  (forall a,b in 0..N:: ((d.a == d.b)  &  (d.a != bot))));
    {- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -}
36) specification:
37)         (exists p, q in 0..N: (p != q) :: (!b.(p)'  &  !b.(q)'  &  (d.(p)' != bot)  &
38)            (d.(q)' != bot)  &  (d.(p)' != d.(q)')  &  f.(p)'  &  f.(q)'))  |
39)         (exists r in 0..N:: (!bg'  &  !b.(r)'  &  f.(r)'  &
                  (d.(r)' != bot)  &  (d.(r)' != dg')));
```

Figure 12.8: The Byzantine agreement problem as input to SYCRAFT.

section, we specify transitions that violate safety by process (and not fault) actions. For instance, it is unacceptable for a process to change its final decision (Line 21). Finally, a non-general process is allowed to read its own and other processes' d values and update its own d and f values (Lines 12-23).

The following states define the invariant predicate: (1) at most one process may be Byzantine (Line 31), (2) the d value of a non-Byzantine non-general process is either 2 or equal to dg (Line 32), and (3) a non-Byzantine undecided process cannot finalize its decision (Line 33), or, if the general is Byzantine and other processes are non-Byzantine their decisions must be identical and not equal to 2 (Line 35).

The safety specification of Byzantine agreement requires agreement and validity. *Agreement* requires that the final decision of two non-Byzantine processes cannot be different (Lines 37-38). And, *validity* requires that if the general is non-Byzantine then the final decision of a non-Byzantine process must be the same as that of the general (Lines 39). Notice that in the presence of a Byzantine process, the program may violate the safety specification.

Similar to other examples, the Byzantine agreement program works fine in the absence of faults. However, it may violate its safety specification (agreement or validity) or deadlock in the presence of faults. Figure 12.9 shows how SYCRAFT adds fault-tolerance to Byzantine agreement. As can be seen the first action is unchanged, i.e., a non-general process can copy the general's decision under no constraints. However, the second action of the input program is revised. Intuitively, the revised action stipulates that a non-general process can finalize its decision only if it agrees with a majority of non-general processes on its decision. Finally, safe recovery actions of a non-general process of the tolerant program resolves deadlock states by reading the decision of other non-general processes in order to adjust its own decision (recovery actions 6 and 7). The process has also the option of finalizing its decision while recovering (recovery actions 8 and 9).

```
---------------------------------------------------------------------------
UNCHANGED ACTIONS:
---------------------------------------------------------------------------
1-((d0==2) & !(f0==1)) & !(b0==1)            -->   (d0 := dg)
---------------------------------------------------------------------------
REVISED ACTIONS:
---------------------------------------------------------------------------
2-(b0==0) & (d0==1) & (d1==1) & (f0==0)      -->   (f0 := 1)
3-(b0==0) & (d0==0) & (d2==0) & (f0==0)      -->   (f0 := 1)
4-(b0==0) & (d0==0) & (d1==0) & (f0==0)      -->   (f0 := 1)
5-(b0==0) & (d0==1) & (d2==1) & (f0==0)      -->   (f0 := 1)
---------------------------------------------------------------------------
NEW RECOVERY ACTIONS:
---------------------------------------------------------------------------
6-(b0==0)& (d0==0)& (d1==1)& (d2==1)& (f0==0) -->   (d0 := 1)
7-(b0==0)& (d0==1)& (d1==0)& (d2==0)& (f0==0) -->   (d0 := 0)
8-(b0==0)& (d0==0)& (d1==1)& (d2==1)& (f0==0) -->   (d0 := 1), (f0 := 1)
9-(b0==0)& (d0==1)& (d1==0)& (d2==0)& (f0==0) -->   (d0 := 0), (f0 := 1)
---------------------------------------------------------------------------
```

Figure 12.9: SYCRAFT output: fault-tolerant Byzantine agreement.

Part IV

Distributed and Parallel Revision Techniques

In Chapter 11, we presented a BDD-based approach to remedy the state explosion problem as well as time complexity in the context of synthesizing distributed programs. Alternatively, in order to overcome the space explosion problem, recently, an increasing interest in parallel and distributed techniques has emerged in the model checking community (e.g., [GMS01, SD97, HGGS00, LSW03, CC06]). Such techniques parallelize *enumerative* or *symbolic* state space of a given model over a network or cluster of workstations and run a distributed verification algorithm over the parallelized state space. On the other hand, the space explosion problem and high time complexity of synthesis methods remain unaddressed.

With this motivation, in this part of the dissertation, we concentrate on distributed and parallel techniques as means to address the state explosion problem and utilizing multiple processors for more efficient program revision. This part is organized as follows. In Chapter 13, we propose two distributed algorithms (multiple processors-distributed memory) for synthesizing failsafe and masking fault-tolerant untimed centralized programs. Then, in Chapter 14, we present a parallel BDD-based algorithm (multiple processors-shared memory) for deadlock resolution. This algorithm can be used as a building block for automated synthesis of masking distributed fault-tolerant programs.

Chapter 13

Distributed Synthesis of Centralized Fault-Tolerant Programs

In this chapter, we concentrate on the problem of designing distributed algorithms (multiple processors-distributed memory) for automated program synthesis. More specifically, we parallelize two synthesis algorithms (from [KA00]) for adding two levels of fault-tolerance, namely failsafe and masking, to existing fault-intolerant programs. As mentioned in previous chapters, intuitively, in the presence of faults, a *failsafe* fault-tolerant program satisfies only its safety specification, but a *masking* fault-tolerant program satisfies both its safety and liveness specifications. We assume that programs are untimed and centralized, where all processes can read and write all program variables in one atomic step. We note that following the Problem Statement 7.3.2, the aforementioned synthesis algorithms *solely* add fault-tolerance to a fault-intolerant program in the sense that they add no new behaviors to the input program in the *absence* of faults.

Similar to distributed model checking techniques, developing distributed synthesis algorithms consists of two phases:

1. parallelizing the state space over a network of workstations, and

2. designing a distributed algorithm that runs on each partition of the state space.

In this chapter, we only focus on the second phase. In particular, we assume that parallelization of state space is already done using one of the known enumerative techniques in the literature. Precisely, we use the state space parallelization technique proposed by Garavel, Mateescu, and Smarandache [GMS01] with some modifications tailored for the purpose of synthesis rather than model checking. Although there exist more efficient ways for parallel construction of state space (e.g., using abstract interpretation), we cannot trivially apply them as a means for synthesizing programs. This is due to the fact that in synthesis (unlike model checking), we usually require full information about the program being synthesized, as we need to manipulate a program by removing or adding computations. Thus, we conservatively choose to develop distributed algorithms that run over the detailed parallelized enumerative state space.

Since the essence of the proposed algorithm in [KA00] for synthesizing failsafe fault-tolerant programs is calculating fixpoint of formulae, in this chapter, we propose a distributed multi-threaded algorithm for calculating smallest and largest fixpoints. Furthermore, since a masking fault-tolerant program recovers to its normal behavior after the occurrence of faults, we also propose a distributed algorithm for synthesizing recovery paths.

The main results of this chapter are as follows. We propose (i) a distributed multi-threaded synthesis algorithm for adding failsafe fault-tolerance, and (ii) a distributed multi-threaded synthesis algorithm for adding masking fault-tolerance to existing fault-intolerant programs. These algorithms involve designing distributed techniques for fixpoint calculations and adding recovery computations to a program. To the best of our knowledge, our method is the first that addresses challenges and proposes solutions for designing distributed algorithms in the context of program synthesis and transformation. We believe that this study paves the way for further research on designing distributed synthesis algorithms.

The rest of this chapter is organized as follows. We first present the state space construction technique due to Garavel, Mateescu, and Smarandache [GMS01] in Section 13.1. We make some modifications to the techniques in [GMS01] tailored for the purpose of synthesis rather than model checking. Then, in Sections 13.2 and 13.3, we present our distributed algorithms for adding failsafe and masking fault-tolerance, respectively.

13.1 Parallel Construction of State Space

In order to represent a program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, Inv_{\mathcal{P}} \rangle$ with state space $\mathcal{S}_{\mathcal{P}}$ on N machines (numbered from 0 to $N-1$), we use the notion of *partitioned* programs. More specifically, the state space $\mathcal{S}_{\mathcal{P}}$ is partitioned to $\mathcal{S}^0_{\mathcal{P}} \cdots \mathcal{S}^{N-1}_{\mathcal{P}}$, where $\mathcal{S}_{\mathcal{P}} = \cup_{i=0}^{N-1} \mathcal{S}^i_{\mathcal{P}}$ and $\mathcal{S}^i_{\mathcal{P}} \cap \mathcal{S}^j_{\mathcal{P}} = \{\}$ for all $0 \leq i \neq j < N$ (i.e., the state space is partitioned into N classes, one class per machine). Likewise, state predicates are partitioned in the same fashion. For instance, machine i contains $Inv^i_{\mathcal{P}}$ and $S^i_{\mathcal{P}}$ partitions of the invariant $Inv_{\mathcal{P}}$ and the fault-span $S_{\mathcal{P}}$. From now on, we call $Inv_{\mathcal{P}}$ the *global invariant* and each $Inv^i_{\mathcal{P}}$ the *local invariant* with respect to machine i. The same concept applies to any other state predicate such as the fault-span $S_{\mathcal{P}}$, i.e., $S_{\mathcal{P}}$ is the global fault-span and $S^i_{\mathcal{P}}$ is the local fault-span with respect to machine i.

The set $T_{\mathcal{P}}$ of transitions is partitioned to $T^0_{\mathcal{P}} \cdots T^{N-1}_{\mathcal{P}}$, where $T_{\mathcal{P}} = \cup_{i=0}^{N-1} T^i_{\mathcal{P}}$, and $(\sigma_0, \sigma_1) \in T^i_{\mathcal{P}}$ iff $(\sigma_0 \in \mathcal{S}^i_{\mathcal{P}} \vee \sigma_1 \in \mathcal{S}^i_{\mathcal{P}})$ for all $0 \leq i < N$ (i.e., if the source and target of a transition belong to different machines, the transition is stored in both the source and target machines). We call such transitions *cross transitions.* Likewise, F and $SPEC_{bt}$ are partitioned in the same fashion. From now on, we call $T_{\mathcal{P}}$ the *global set of program transitions* and each $T^i_{\mathcal{P}}$ the *local set of program transitions* with respect to machine i. The same concept applies to any other set of transitions such as the set of faults F and the set of bad transitions $SPEC_{bt}$.

Remark 13.1.1 We choose to store cross transitions in both source and target machines due to two reasons:

1. as we shall see in Sections 13.2 and 13.3, such duplication decreases the number of potential broadcast messages considerably, and

2. it allows us to efficiently do both forward and backward reachability analysis at the same time.

In fact, this deviation from distributed model checking techniques is due to the nature synthesis as compared to verification.

Assumption 13.1.2 In our synthesis algorithms, we assume that the input fault-intolerant program $\mathcal{P}$ is already partitioned over a network using a reasonable static partition function

$h : \mathcal{S}_{\mathcal{P}} \rightarrow [0, N-1]$ using the above parallelization method. In other words, machine i contains a state σ iff $h(\sigma) = i$. We also assume that all the synthesis processes over the network have a replica of h.

Revised problem statement

With this setting, we revise the Problem Statement 7.3.2 as follows. Given are a partition function h, a partitioned program $T^0_{\mathcal{P}} \cdots T^{N-1}_{\mathcal{P}}$ with state space $\mathcal{S}^0_{\mathcal{P}} \cdots \mathcal{S}^{N-1}_{\mathcal{P}}$, local invariants $Inv^0_{\mathcal{P}} \cdots Inv^{N-1}_{\mathcal{P}}$, a partitioned class of faults $F^0 \cdots F^{N-1}$, and safety specification $SPEC^0_{bt} \cdots SPEC^{N-1}_{bt}$ such that $\mathcal{P} \models_{Inv_{\mathcal{P}}} SPEC_{\overline{bt}}$. Our goal is to design distributed algorithms that synthesize a program $\mathcal{P}'$ with invariant $Inv_{\mathcal{P}'}$ such that $\mathcal{P}'$ is failsafe/masking F-tolerant to $SPEC_{\overline{bt}}$ from $Inv_{\mathcal{P}'}$.

13.2 Distributed Addition of Failsafe Fault-Tolerance

In order to synthesize a failsafe fault-tolerant program, we transform $\mathcal{P}$ into $\mathcal{P}'$ such that transitions of $SPEC_{bt}$ occur in no computation prefixes of $\mathcal{P}'$. Towards this end, we parallelize the proposed centralized algorithm in [KA00] for adding failsafe fault-tolerance.

Algorithm sketch. The essence of adding failsafe fault-tolerance consists of two parts: (1) a smallest fixpoint calculation for identifying the set of states from where safety may be violated, and (2) a largest fixpoint calculation for computing the invariant of the failsafe program. Our algorithm consists of a set of distributed processes each running on one machine across the network. Each process consists of two threads, namely, Distributed_Add_failsafe (cf. Thread 13.1) and MessageHandler (cf. Thread 13.5). Briefly, the thread Distributed_Add_failsafe is in charge of initiating local fixpoint calculations and managing synchronization points of the algorithm. The thread MessageHandler is responsible for handling messages sent by other synthesis processes across the network and invoking appropriate procedures. The thread Distributed_Add_failsafe consists of three main parts, namely, Lines 1-4 which is a smallest fixpoint computation, Lines 5-9 which is a largest fixpoint computation, and Lines 10-13 where we check the emptiness of the synthesized program (to declare failure or success). It also invokes three procedures, namely, FindLocalUnsafeStates, RemoveLocalDeadlocks, and EnsureClosure.

Thread 13.1 Distributed_Add_failsafe

Input: local program transitions $T^i_{\mathcal{P}}$, invariant $Inv^i_{\mathcal{P}}$, fault transitions F^i, safety specification $SPEC^i_{bt}$, state space $\mathcal{S}^i_{\mathcal{P}}$, **integer** N, partition function h, and **Boolean** $bLeader^i$.
Output: a failsafe partition of program $\mathcal{P}$.

1: $cbSnt^i,\ cbRcvd^i := 0;\ ns^i := \{\}$;
2: $ms^i := \{\sigma_0 \mid \exists \sigma_1 \in \mathcal{S}^i_{\mathcal{P}} : (\sigma_0, \sigma_1) \in F^i\ \wedge\ (\sigma_0, \sigma_1) \in SPEC^i_{bt}\}$;
3: $ms^i :=$ FindLocalUnsafeStates$(\mathcal{S}^i_{\mathcal{P}}, ms^i, F^i)$;
4: BlkReceive (`Trm_dtct`); $\{\longleftarrow\}$
5: $cbSnt^i,\ cbRcvd^i := 0$;
6: $mt^i := \{(\sigma_0, \sigma_1) \mid \sigma_1 \in (ms^i \cup ns^i)\ \vee\ (\sigma_0, \sigma_1) \in SPEC^i_{bt}\}$;
7: $Inv^i_{\mathcal{P}} := Inv^i_{\mathcal{P}} - ms^i;\ T^i_{\mathcal{P}} := T^i_{\mathcal{P}} - mt^i$;
8: $Inv^i_{\mathcal{P}'}, T^i_{\mathcal{P}'} :=$ RemoveLocalDeadlocks$(Inv^i_{\mathcal{P}}, T^i_{\mathcal{P}})$;
9: BlkReceive (`Trm_dtct`); $\{\longleftarrow\}$
10: **if** $(Inv^i_{\mathcal{P}'} \neq \{\})$ **then**
11: $\quad T_{\mathcal{P}'} := \cup_{i=0}^{N-1} T^i_{\mathcal{P}'}$;
12: $\quad Inv_{\mathcal{P}'} := \cup_{i=0}^{N-1} Inv^i_{\mathcal{P}'}$;
13: $\quad$ **return** $T_{\mathcal{P}'},\ Inv_{\mathcal{P}'}$;
14: **else**
15: $\quad$ **if** $(bLeader^i)$ **then**
16: $\quad\quad$ Send$((i+1) \mod N$, `Empt_inv`$(0))$;
17: $\quad$ **end if**
18: **end if**

Assumption 13.2.1 Throughout this chapter, we assume that procedure invocations are *atomic*.

We now describe our algorithm in detail. First, the thread Distributed_Add_failsafe finds the set ms^i of states from where a single fault transition violates the safety (Line 2). Next, we invoke the procedure FindLocalUnsafeStates where we find the set of states from where faults alone may violate the safety (Line 3). We find this set by calculating the smallest fixpoint of backward reachable states, given the initial set ms^i (see Procedure 13.2). In this calculation, if we find a fault transition, say (σ_0, σ_1), where $\sigma_1 \in ms^i$, but σ_0 resides in a machine other than i (i.e., $h(\sigma_0) \neq i$), we send a `New_ms` message to process $h(\sigma_0)$ indicating that σ_0 is a state from where faults alone may violate the safety specification.

Notation: At the receiver's side, we denote messages by $msg_j(params)$, where msg is the name of message, j is the sender process, and *params* is a list of parameters sent along with the message. All messages (except `Trm_dtct`) are handled in the thread MessageHandler. At the sender's side, we omit the sender's subscript.

The receiver of a `New_ms` message (cf. Lines 3-7 in Thread 13.5) adds σ_0 to its local ms^i

(Line 4) and invokes the procedure FindLocalUnsafeStates (Line 6) so that by taking σ_0 into account, new states from where faults alone may violate the safety specification are explored. The set ns^i consists of states that are in ms^j. Notice that every time a process sends (respectively, receives) such messages, it increments the variable $cbSnt^i$ (respectively, $cbRcvd^i$). We shall use these variables for termination detection as a means to synchronize processes at certain points.

Procedure 13.2 FindLocalUnsafeStates

Input: state predicates $\mathcal{S}^i$, ms^i and transition predicate F^i.
Output: returns the set of states from where safety may be violated by faults alone.

while $(\exists \sigma_0, \sigma_1 : (\sigma_1 \in ms^i \;\wedge\; (\sigma_0, \sigma_1) \in F^i))$ **do**
  if $h(\sigma_0) = i$ **then**
    $ms^i := ms^i \cup \{\sigma_0\}$;
  else
    $\textsc{Send}(h(\sigma_0), \texttt{New\_ms}(\sigma_0, \sigma_1))$; $cbSnt^i := cbSnt^i + 1$;
  end if
end while
return ms^i;

The next phase of the algorithm is removing the states of global ms from the global invariant. To this end, we need to have a synchronization mechanism to ensure that calculation of ms^i is completed for all $i \in [0..N-1]$. In particular, we use the termination detection technique proposed by Mattern [Mat87]. More specifically, in Line 6, the thread Distributed_Add_failsafe waits to receive a `Trm_dtct` message indicating that all processes are finished by calculating their local ms^i and all communication channels are empty. The arrows ($\longleftarrow$) in Thread 13.1 mark the synchronization barriers. We will describe the termination detection technique later in this subsection.

Procedure 13.3 RemoveLocalDeadlocks

Input: state predicate Inv^i and transition predicate T^i.
Output: returns the largest subset of Inv^i such that computations of T^i within that subset are infinite.

while $(\exists \sigma_1 \in Inv^i_{\mathcal{P}} : (\forall \sigma_2 \mid (\exists \sigma_0 \mid (\sigma_0, \sigma_2) \in T^i) \;:\; (\sigma_1, \sigma_2) \notin T^i))$ **do**
  $Inv^i := Inv^i - \{\sigma_1\}$;
  $T^i :=$ EnsureClosure(T^i, Inv^i, σ_1);
end while
return Inv^i, T^i;

Procedure 13.4 EnsureClosure

Input: state predicate Inv^i, transition predicate T^i, and state σ_1.
Output: state predicate Inv^i and transition predicate T^i where Inv^i is closed in T^i.

1: **while** $(\exists \sigma_0 \ : \ ((\sigma_0, \sigma_1) \in T^i \ \wedge \ h(\sigma_0) \neq i))$ **do**
2:     $\textsc{Send}(h(\sigma_0), \texttt{New\_ds}(\sigma_0, \sigma_1))$;
3:     $cbSnt^i := cbSnt^i + 1$;
4:     $T^i := T^i - \{(\sigma_0, \sigma_1)\}$;
5: **end while**
6: **return** $T^i - \{(\sigma_0, \sigma_1) \mid \sigma_0 \in Inv^i_{\mathcal{P}}\}$;

After calculating the global set ms, we remove this set from the invariant to ensure that no computation of $\mathcal{P}'$ that starts from a state in $Inv_{\mathcal{P}'}$ violates the safety specification. We also remove the transitions of the set mt^i from $T^i_{\mathcal{P}}$, where mt^i consists of transitions whose target states are in ms^i or directly violate the safety specification (Line 7). Notice that this removal may create *deadlock* states (i.e., states from where there exist no outgoing transitions). Thus, the thread Distributed_Add_failsafe invokes the procedure RemoveLocalDeadlocks (Line 8) to remove deadlock states which is in turn calculating the largest fixpoint of backward reachable states, given the initial set $Inv^i_{\mathcal{P}}$. In other words, it keeps removing deadlock states until it reaches a fixpoint (see Procedure 13.3). In this calculation, since removal of a deadlock state, say σ_1, may create transitions, say (σ_0, σ_1), such that (σ_0, σ_1) violates the closure of invariant, we invoke the procedure EnsureClosure to ensure that no such transitions exist in the final synthesized program. Furthermore, if we encounter a program transition, say (σ_0, σ_1), where σ_1 is a deadlock state and σ_0 resides in a machine other than i (i.e., $h(\sigma_0) \neq i$), then we send a `New_ds` message to process $h(\sigma_0)$ indicating that σ_0 *might* be a deadlock state (Line 2 in Procedure 13.4). Upon receipt of such a message (cf. Line 8 in Figure 13.5), the receiver removes the transition (σ_0, σ_1) to maintain consistency of transitions and then invokes the procedure RemoveLocalDeadlocks (cf. Line 10 in Figure 13.5) to remove possible new deadlock states due to removal of (σ_0, σ_1). Similar to the calculation of ms, our algorithm ensures completion of calculation of the largest fixpoint $Inv_{\mathcal{P}'}$ using the same termination detection technique (Line 9 in Figure 13.1).

At this point, each process has synthesized a local set of program transitions $T^i_{\mathcal{P}'}$ with a local invariant $Inv^i_{\mathcal{P}'}$. The union of these portions is the final synthesized program, i.e., $T_{\mathcal{P}'} = \cup_{i=0}^{N-1} T^i_{\mathcal{P}'}$ and $Inv_{\mathcal{P}'} = \cup_{i=0}^{N-1} Inv^i_{\mathcal{P}'}$. However, since invariant predicates cannot be empty, if $Inv_{\mathcal{P}'}$ turns out to be equal to the empty set, the algorithm declares failure. To test the emptiness of $Inv_{\mathcal{P}'}$, a pre-specified *leader* process identified by the variable *bLeader* initiates an emptiness polling of the global invariant $Inv_{\mathcal{P}'}$ as follows. For this polling (and also termination detection), we consider a unidirectional virtual ring which connects every machine i to its successor machine $(i+1) \mod N$. Note that this virtual ring is independent of the fully connected topology of the network. Now, if the local invariant of the leader is equal to the empty set then it sends an `Empt_inv`(0) message to its first neighbor on the virtual ring (process $(i+1) \mod N$) indicating that its own local invariant is equal to the empty set (cf. Lines 15-17 in Thread 13.1). If the local invariant of the $((i+1) \mod N)^{\text{th}}$ process is equal to the empty set as well, it increments the value of k (the integer received along with the message `Empt_inv`) by one and sends the same message to the next process on the ring (cf. Line 14 in Figure 13.5). Otherwise, it does not change the value of k and sends an `Empt_inv`(k) message to the next process (Line 18). Upon the completion of one round of sending the `Empt_inv` messages, the leader finally finds out whether the global invariant $Inv_{\mathcal{P}'}$ is equal to the empty set or not (Lines 22-28). If the global invariant $Inv_{\mathcal{P}'}$ is indeed equal to the empty set then the leader declares failure (Line 22). Otherwise, it calculates and returns $\mathcal{P}'$ and $Inv_{\mathcal{P}'}$ (Line 25). Notice that Lines 11-13 in Thread 13.1 and 25 and in thread 13.5 respectively *describes* that the output of the distributed algorithm is indeed a program which is the union of all local sets of transitions and local invariants.

Termination detection

In order to detect the termination of the fixpoint calculations, we use a virtual ring-based algorithm inspired by Mattern [Mat87]. According to the general definition, global termination is reached when all local computations are complete (i.e., each machine i has calculated a local fixpoint) and all communication channels are empty (i.e., all sent transitions have been received). The core of the termination detection algorithm is as follows. Every time the leader process finishes its local fixpoint calculations, it checks whether global termination has been reached by generating two successive waves of `Report_rcv` (respectively, `Report_snd`) messages on the virtual ring to collect the number of messages received (respectively, sent) by all machines. A message $\texttt{Report\_rcv}_j(k)$ (respectively, $\texttt{Report\_snd}_j(k)$) received by machine i indicates that k messages have been received (respectively, sent) by the machines from the leader up to $j = (i-1) \mod N$. Each machine i counts the messages it has received and sent using two integer variables $cbRcvd^i$ and $cbSnt^i$, and adds

Thread 13.5 MessageHandler (part 1)

1: $msg :=$ RECEIVE();
2: **Switch** msg **is**
3: **Case** $\texttt{New\_ms}_j(\sigma_0, \sigma_1)$:
4: $ms^i := ms^i \cup \{\sigma_0\}$; $ns^i := ns^i \cup \{\sigma_1\}$;
5: $cbRcvd^i := cbRcvd^i + 1$;
6: $ms^i :=$ FindLocalUnsafeStates($\mathcal{S}^i_{\mathcal{P}}, ms^i, F^i$);
7: **EndCase**
8: **Case** $\texttt{New\_ds}_j(\sigma_0, \sigma_1)$:
9: $T^i_{\mathcal{P}} := T^i_{\mathcal{P}} - \{(\sigma_0, \sigma_1)\}$; $cbRcvd^i := cbRcvd^i + 1$;
10: $Inv^i_{\mathcal{P}'}, T^i_{\mathcal{P}'} :=$ RemoveLocalDeadlocks($Inv^i_{\mathcal{P}}, T^i_{\mathcal{P}}$);
11: **EndCase**
12: **Case** $\texttt{Empt\_inv}_j(k)$:
13: **if** $(\neg bLeader \ \wedge \ Inv^i_{\mathcal{P}} = \{\})$ **then**
14: SEND($(i+1) \mod N$, $\texttt{Empt\_inv}(k+1)$);
15: **return** $\{\}$;
16: **else**
17: **if** $(\neg bLeader \ \wedge \ Inv^i_{\mathcal{P}} \neq \{\})$ **then**
18: SEND($(i+1) \mod N$, $\texttt{Empt\_inv}(k)$);
19: **return** $T^i_{\mathcal{P}'}$, $Inv^i_{\mathcal{P}'}$;
20: **else**
21: **if** $(bLeader \ \wedge \ (k = N-1))$ **then**
22: `declare no failsafe program` $\mathcal{P}'$ `exists`;
23: **exit**();
24: **else**
25: **return** $T_{\mathcal{P}'} := \cup_{i=0}^{N-1} T^i_{\mathcal{P}'}, Inv_{\mathcal{P}'} := \cup_{i=0}^{N-1} Inv^i_{\mathcal{P}'}$;
26: **end if**
27: **end if**
28: **end if**
29: **EndCase**
30: **Case** $\texttt{Report\_rcv}_j(k)$:
31: **if** $(\neg bLeader^i)$ **then**
32: SEND($(i+1) \mod N$, $\texttt{Report\_rcv}(k + cbRcvd^i)$);
33: **else**
34: $nbTotal := k$;
35: SEND($(i+1) \mod N$, $\texttt{Report\_snd}(cbSnt)^i$);
36: **end if**
37: **EndCase**
38: **Case** $\texttt{Report\_snd}_j(k)$:
39: **if** $(\neg bLeader^i)$ **then**
40: SEND($(i+1) \mod N$, $\texttt{Report\_snd}(k + cbSnt^i)$);
41: **else**
42: **if** $(nbTotal = k)$ **then**
43: SEND($[(i+1) \mod N..(i+N-1) \mod N]$, $\texttt{Trm\_dtct}$);
44: **end if**
45: **end if**
46: **EndCase**

their values to the numbers carried by `Report_rcv` and `Report_snd` messages (Lines 32, 35, and 40). Upon receipt of the $\texttt{Report\_snd}_j(k)$ message ending the second wave, the leader

Thread 13.6 MessageHandler (part 2)

Case `New_fs`(σ_0):
  $S_1^i := S_1^i - \{\sigma_0\}$;
  $cbRcvd^i := cbRcvd^i + 1$;
  $S_1^i :=$ConstructLocalFaultSpan$(S_1^i, S_2^i - S_1^i, F^i)$;
EndCase
Case `Search_path`$_j(X)$:
  $r^i := \{\}$; $cbRcvd^i := cbRcvd^i + 1$;
  for each $\sigma_0 \in X$ **do**
    if $(\exists \sigma_1 \ : \ (\text{Rank}(\sigma_1) \neq \infty)) \ \wedge ((\sigma_0, \sigma_1) \notin mt^i)$ **then**
      $r^i := r^i \cup \{(\sigma_0, \sigma_1, \text{Rank}(\sigma_1) + 1)\}$;
    end if
  end for
  SEND$(j$, `New_path`$(r^i))$; $cbSnt^i := cbSnt^i + 1$;
EndCase
Case `New_path`$_j(r^i)$:
  $q^i := \{\}$;
  $cbRcvd^i := cbRcvd^i + 1$;
  for each(σ_0, σ_1, a) s.t. $((\sigma_0, \sigma_1, a) \in r^i \ \wedge \ \sigma_0 \in (S_2^i - S_1^i))$ **do**
    if $(\sigma_0, \sigma_1) \notin T_{\mathcal{P}}^i$ **then**
      $T_{\mathcal{P}}^i := T_{\mathcal{P}}^i \cup (\sigma_0, \sigma_1)$; $q^i := q^i \cup (\sigma_0, \sigma_1)$;
      Rank$(\sigma_0) := a$;
      $S_1^i, S_2^i := S_1^i \cup \{\sigma_0\}, S_2^i - \{\sigma_0\}$;
      $T_1^i, T_1^i :=$ ConstructLocalRecoveryPaths$(Inv_1^i, S_2^i, T_1^i, mt^i)$;
      SEND$(j$, `Confirm_trns`$(q^i))$; $cbSnt^i := cbSnt^i + 1$;
    end if
  end for
EndCase
Case `Confirm_trns`$_j(q^i)$:
  $T_{\mathcal{P}}^i := T_{\mathcal{P}}^i \cup q^i$;
  $cbRcvd^i := cbRcvd^i + 1$;
  SEND$(j,$`Commit`$)$;
  $cbSnt^i := cbSnt^i + 1$;
EndCase
Case `Commit`$_j$:
  $cbRcvd^i := cbRcvd^i + 1$;
  Wait to receive `Commit` message from all providers;
  SEND$(i + 1 \mod N$, `Token`$)$;
  $cbSnt^i := cbSnt^i + 1$;
EndCase
Case `Token`$_j$:
  $cbRcvd^i := cbRcvd^i + 1$;
  SEND$([(i + 1) \mod N..(i + N - 1) \mod N]$, `Search_path`$(S_2^i - S_1^i))$;
  $cbSnt^i := cbSnt^i + 1$;
EndCase
EndSwitch

machine checks whether the total number k of messages sent is equal to the total number $nbTotal$ of messages received (the result of the `Report_rcv` wave). If this is the case, it informs the other machines that termination has been reached, by sending a broadcast

Trm_dtct message. Otherwise, the leader concludes that termination has not been reached yet and will generate a new termination detection wave later (Line 43).

Theorem 13.2.2 *The algorithm Distributed_Add_failsafe is sound and complete.*

Proof. Since the output of our algorithm is identical to the output of the centralized algorithm by Kulkarni and Arora [KA00], the proof of soundness and completeness immediately follows. ∎

Performance of parallelized addition of failsafe

Distributing the synthesis algorithm is aimed at reducing the space complexity and time complexity. Of these, similar to the goals for distributed model checking, reducing the space complexity is a higher priority. We expect that our approach would assist in this case. In particular, if N machines are used to perform synthesis then each of them is expected to have $(1/N)^{\text{th}}$ number of states and at most $(2/N)^{\text{th}}$ number of transitions (because a transition may be stored in up to two machines). Regarding time complexity, in each phase, a machine performs some local computation that results a set of queries (e.g., New_ms, New_ds, etc.) for other machines. Now, consider the role of the two threads Distributed_Add_failsafe and MessageHandler. The thread MessageHandler provides a new list of tasks (received from other machines) that should be performed by Distributed_Add_failsafe. Since Distributed_Add_failsafe begins with a list of tasks (based on its local states and transitions) and MessageHandler continues to provide new tasks based on requests received from others, we expect that the list of tasks that Distributed_Add_failsafe needs to perform will typically be nonempty at all times. In other words, communication cost will not be in the critical path for the synthesis. Therefore, we expect that the distributed synthesis algorithm will be able to provide significant benefits regarding time complexity as well.

13.3 Distributed Addition of Masking Fault-Tolerance

In order to synthesize a masking program, we should generate a program $\mathcal{P}'$ with invariant $Inv_{\mathcal{P}'}$ and fault-span $S_{\mathcal{P}'}$, such that $\mathcal{P}'$ never violates its safety specification and if faults perturb the state of $\mathcal{P}'$ to a state in $S_{\mathcal{P}'}$, it recovers to $Inv_{\mathcal{P}'}$ within a finite number of recovery steps. Similar to the distributed algorithm for adding failsafe fault-tolerance, our algorithm for adding masking fault-tolerance consists of two threads Distributed_Add_masking (cf. Thread 13.7) and MessageHandler (cf. Thread 13.5).

Our first estimate of a masking program is a failsafe program. Hence, we let our first estimate Inv_1^i be the local invariant of its failsafe fault-tolerant program (cf. Lines 1-2 in Figure 13.7). Likewise, we estimate the local fault-span to be S_1^i where S_1^i includes all the states in the local state space minus the states from where safety of $\mathcal{P}'$ may be violated (Lines 3-4). Next, we compute the local set of transitions T_1^i, local fault-span S_1^i, and local invariant Inv_1^i in a loop (Lines 6-25). This loop consists of three main steps for constructing recovery paths, calculating fault-span, and calculating invariant as follows:

1. In order to compute the local set of transitions T_1^i, we construct recovery paths from each state in the fault-span to a state in the invariant. To this end, we identify two types of recovery paths: (1) recovery paths consist of only local program transitions, and (2) recovery paths consist of both local program transitions as well as cross transitions. We note that since these transitions originate outside the invariant, they do not violate the second constraint of the problem statement (i.e., in the absence of faults, no new computation is introduced to fault-tolerant program).

   Recovery paths through local transitions. The thread Distributed_Add_masking invokes the procedure ConstructLocalRecoveryPaths (Line 9), which identifies layers of states in the local fault-span corresponding to the number of steps of recovery paths, in a loop (Lines 4-10 in Procedure 13.8). In the beginning of the loop it assigns a rank to each state which is equal to the number of recovery steps from that state to a state in the local invariant. In this setting, the rank of states in the local invariant are zero. In the first iteration of the loop, we identify the set of states from where one-step recovery to the local invariant is possible while maintaining the safety, i.e., $X_1^i = \{\sigma_0 \mid \sigma_0 \in (S_1^i - Inv_1^i) \ \wedge \ \exists \sigma_1 \in Inv_1^i \ : \ (\sigma_0, \sigma_1) \notin mt^i\}$. Thus, we add the transitions, say (σ_0, σ_1) where $\sigma_0 \in X_1^i$ and $\sigma_1 \in Inv_{\mathcal{P}}^i$, to the set of local program transitions. In the second iteration of the loop, we identify the set of states from where two-step recovery is possible. Indeed, this is equivalent to identifying the set of states from where one-step recovery is possible from $S_1^i - X_1^i$ to the set $X_1^i \cup Inv_1^i$. Continuing thus inductively, we identify layers of states from where multi-step recovery is possible. Finally, we reach a point where we identify the set X_1^i of states from where recovery to the local invariant using local transitions is possible and the set $S_1^i - X_1^i$ of states

Thread 13.7 Distributed_Add_masking

Input: local program transitions $T^i_{\mathcal{P}}$, invariant $Inv^i_{\mathcal{P}}$, fault transitions F^i, safety specification $SPEC^i_{bt}$, state space $\mathcal{S}^i_{\mathcal{P}}$, **integer** N, partition function h, and **Boolean** $bLeader^i$.
Output: a masking partition of program $\mathcal{P}$.

Compute ms^i and mt^i as in Distributed_Add_failsafe;
Let Inv^i_1 be the local invariant of the failsafe version of $\mathcal{P}$;
$S^i_1 := \mathcal{S}^i_{\mathcal{P}} - ms^i$;
$\forall \sigma \in (S^i_1 - Inv^i_1) \; : \; \mathrm{Rank}(\sigma) := \infty$;
$T^i_1 := T^i_{\mathcal{P}}$;
repeat
  $S^i_2, Inv^i_2 := S^i_1, Inv^i_1$;
  $cbSnt^i$, $cbRcvd^i := 0$;
  $T^i_1, S^1_1 :=$ ConstructLocalRecoveryPaths$(Inv^i_1, S^1_1, T^i_{\mathcal{P}}, mt^i)$;
  BlkReceive (`Trm_dtct`); $\{\longleftarrow\}$
  $cbSnt^i$, $cbRcvd^i := 0$;
  if $(bLeader)$ **then**
    Send$([(i+1) \mod N..(i+N-1) \mod N]$, `Search_path`$(S^i_1 - S^i_1))$;
    $cbSnt^i := cbSnt^i + 1$;
  end if
  BlkReceive (`Trm_dtct`); $\{\longleftarrow\}$
  $cbSnt^i$, $cbRcvd^i := 0$;
  $S^i_1 :=$ ConstructLocalFaultSpan$(S^i_1, S^i_2 - S^i_1, F^i)$;
  BlkReceive (`Trm_dtct`); $\{\longleftarrow\}$
  $cbSnt^i$, $cbRcvd^i := 0$;
  $Inv^i_1 :=$ RemoveLocalDeadlocks$(Inv^i_1 \cap S^i_1, T^i_1)$;
  BlkReceive (`Trm_dtct`); $\{\longleftarrow\}$
  if $(Inv^i_1 = \{\} \; \vee \; S^i_1 = \{\})$ **then break**;
  end if
until $(S^i_1 = S^i_2 \; \wedge \; Inv^i_1 = Inv^i_2)$
BlkReceive (`Trm_dtct`); $\{\longleftarrow\}$
$S^i_{\mathcal{P}'}, Inv^i_{\mathcal{P}'} := S^i_1, Inv^i_1$;
if $(bLeader^i \; \wedge \; (Inv^i_{\mathcal{P}'} = \{\}))$ **then**
  Send$((i+1) \mod N$, `Empt_inv`$(0))$;
end if
if $(bLeader^i \; \wedge \; (S^i_{\mathcal{P}'} = \{\}))$ **then**
  Send$((i+1) \mod N$, `Empt_fs`$(0))$;
end if

from where such recovery is not possible.

Recovery paths through cross transitions. After constructing local recovery paths, the leader process initiates a wave of communication among all processes to identify the set of states from where local recovery is not possible, but recovery through cross transitions is possible. More specifically, the leader process sends a `Search_path`

Procedure 13.8 ConstructLocalRecoveryPaths

Input: state predicates Inv^i, S^i and transition predicates T^i, mt^i.
Output: returns transition predicate containing recovery transitions

$X_1^i := Inv_{\mathcal{P}}^i$;
$j = 0$;
$X_2^i := \{\}$;
repeat
  $\forall \sigma \in (X_1^i - X_2^i) \;:\; \text{Rank}(\sigma) := j$;
  $X_2^i := X_1^i;\ j := j+1$;
  $r^i := \{(\sigma_0, \sigma_1) \mid \sigma_0 \in (S_1^i - X_1^i) \;\wedge\; \sigma_1 \in X_1^i\} - mt^i$;
  $T^i := T^i \mid Inv^i \cup r^i$;
  $X_1^i := X_1^i \cup \{\sigma_0 \mid \exists \sigma_1 \;:\; (\sigma_0, \sigma_1) \in r_i\}$;
until $(X_1^i = X_2^i)$
return T^i, X^i;

message to all other processes (Line 13 in Thread 13.7). Let us call the process which sends a `Search_path` the *requester* process. Upon receipt of this message along with the set X of states from where local recovery is not possible (Line 52 in Thread 13.5), each process offers a recovery cross transition, say (σ_0, σ_1), provided $(\sigma_0, \sigma_1) \notin mt^i$ and there exists a recovery path from σ_1 (i.e., $\text{Rank}(\sigma_1) \neq \infty$), for each state $\sigma_0 \in X$ (Line 56). Let us call such processes the *providers*. Each provider sends a `New_path` message carrying the set r^i of cross recovery transitions along with the rank of state σ_0 to the requester (Line 59). Obviously, if the requester accepts the provider's transitions, the rank of σ_0 will be $\text{Rank}(\sigma_1) + 1$.

Upon receipt of this message (Line 61 in Thread 13.5), the requester adds the new recovery cross transitions, say (σ_0, σ_1), to its set of local program transitions (Line 66) and sets the rank of source states σ_0 (Line 67). These states should be added to the local fault-span (Line 68). Next, it invokes the procedure ConstructLocalRecoveryPaths to add new possible local recovery transitions by taking the newly added recovery cross transitions into account (Line 69). Then, it sends a `Confirm_trns` message to the providers of the cross transitions so that the set of cross transitions of providers and the requester processes are consistent (Line 70). Obviously, if the requester receives other offers for a cross transition originated at σ_0, say (σ_0, σ_1) with rank a, where the current rank of σ_0 is greater than a, then the requester can replace its current cross transition with (σ_0, σ_1). However, we do not illustrate such implementation details in

Procedure 13.9 ConstructLocalFaultSpan

Input: state predicates S_1^i, S_2^i and transition predicatef F^i.
Output: returns the largest fault-span closed in F^i

while $(\exists \sigma_0, \sigma_1 \ : \ ((\sigma_0 \in S_1^i) \ \wedge \ (\sigma_1 \in S_2^i) \ \wedge \ (\sigma_0, \sigma_1) \in F^i))$ **do**
  $S_1^i := S_1^i - \{\sigma_0\}$;
end while
for each $\sigma_1 \in S_2^i$ **do**
  if $(\exists \sigma_0 \ : \ ((\sigma_0, \sigma_1) \in F^i \ \wedge \ h(\sigma_0) \neq i))$ **then**
    $\textsc{Send}(h(\sigma_0), \texttt{New\_fs}(\sigma_0))$;
    $cbSnt^i := cbSnt^i + 1$;
  end if
end for
return S_1^i;

the algorithms.

Finally, upon the receipt of a `Confirm_trns` message, the providers add the set q^i of cross transitions (selected by the requester) to their set of local program transitions as well (Line 74). At this point, providers send a `Commit` message to the requester (Line 77) indicating that the changes are committed. Upon receipt of `Commit` message from all providers (Lines 81-82), the requester sends a `Token` message to the next process on the virtual ring (Line 83) so that it starts identifying the cross recovery transitions in the same fashion (Line 88). We continue doing this until no cross transition is added across the network.

Notice that in both types of recovery paths, we do not introduce cycles to the fault-span, as we do not add transitions from a state with a lower rank to a state with higher rank. Hence, after occurrence of faults, recovery within a finite number of steps is guaranteed. We synchronize the completion of construction of recovery paths in Line 10.

2. Since there may exist states from where recovery to the invariant is not possible, we need to recompute the local fault-span by removing the states from where closure of fault-span is violated through fault transitions. To this end, we invoke the procedure ConstructFaultspan which is a largest fixpoint calculation (Line 18 in Thread 13.7) to calculate the largest fault-span which is closed in $\mathcal{P}[]f$. Since this removal may cause other states in the local fault-span of other processes to violate the closure of

the global fault-span, we send a `New_fs` message to such processes to indicate this fact (Line 6 in Procedure 13.9). Note that in order to synchronize the completion of calculation of local fault-spans, here as well, we need a barrier synchronization (Line 19).

3. Due to the removal of some states in step 2, we recompute the local invariant by invoking the procedure RemoveLocalDeadlocks. Notice that since Inv_1^i must be a subset of S_1^i, this invocation is parameterized by $Inv_1^i \cap S_1^i$ (Line 21). At this point, if both Inv_1^i and S_1^i are nonempty, we jump back to step 1 and we keep repeating the loop until a fixpoint is reached, i.e., $(S_1^i = S_2^i \wedge Inv_1^i = Inv_2^i)$.

Upon the termination of the repeat-until loop, recovery without violation of the safety specification from S_1^i to Inv_1^i is provided. At this point, if there exist processes i and j such that $Inv_{\mathcal{P}'}^i$ and $S_{\mathcal{P}'}^j$ are both nonempty then we have a solution to the synthesis problem. Thus, similar to addition of failsafe, we run an emptiness poll among the processes (Lines 28-33). To this end, we send a `Empt_fs`(0), which is similar to `Empt_inv`, except the message handler tests the emptiness of the local fault-span rather than local invariant. We skip including this message in Thread 13.5.

We note that, in this chapter, we have modified the recovery mechanism of the centralized algorithm in [KA00]. This is due to the fact that in that algorithm the authors add all possible transitions, i.e., the set $T^1|Inv_1 \cup \{(\sigma_0, \sigma_1) \mid \sigma_0 \in S_1 - Inv_1 \ \wedge \ \sigma_1 \in S_1\}$, and then remove non-progress cycles. However, since the size of this set in worst case is in the square order of the size of the state space, it implies that in worst case, each machine i must store a set whose size is in the square order of the state space which obviously does not make sense. Hence, instead of adding all possible transitions and removing cycles, we construct recovery paths in a more space-efficient way in a stepwise manner using the notion of layered fault-span (cf. the Procedure ConstructLocalRecoveryPaths).

Theorem 13.3.1 *The algorithm Distributed_Add_masking is sound.* ∎

Chapter 14

Parallelizing Symbolic Deadlock Resolution

In Chapter 11, we observed that depending upon the structure of the given distributed intolerant program, performance of synthesis may suffer from several major complexity obstacles, namely *generation of fault-span*, *resolution of deadlock states*, *cycle detection*, and *addition of recovery*. Thus, more efficient techniques are still needed to overcome the aforementioned bottlenecks. In this chapter, we focus on parallelizing deadlock resolution in symbolic synthesis of fault-tolerant distributed programs. Deadlock resolution is especially crucial in the context of dependable systems, as it guarantees that the synthesized fault-tolerant program meets its liveness requirements even in the presence of faults.

This chapter is organized as follows. First, in Section 14.1, we describe the problem in detail. Then, in Section 14.2, we present our parallel algorithm for deadlock resolution.

14.1 The Deadlock Resolution Problem

We now describe the issue of deadlock resolution using the *Byzantine agreement* (denoted $\mathcal{BA}$) problem [LSP82]. We omit other steps involved in synthesizing a fault-tolerant version of $\mathcal{BA}$ (e.g., fault-span generation, preserving safety, and reconstructing invariant predicate), as they are not in the scope of this paper. $\mathcal{BA}$ consists of a *general*, say g, and three (or more) *non-general* processes: j, k, and l. Each process of $\mathcal{BA}$ maintains a decision d; for the general, the decision can be either 0 or 1, and for the non-general processes, the decision can be 0, 1, or $\perp$, where the value $\perp$ denotes that the corresponding process has not yet received the decision from the general. Each non-general process also maintains a Boolean variable f that denotes whether that process has finalized its decision. For each process, a Boolean variable b shows whether or not the process is Byzantine. In the *fault-intolerant* version of this program, each non-general process copies the decision from the general and then finalizes (outputs) that decision, provided it is non-Byzantine. A fault transition can cause a process to become Byzantine, if no other process is initially Byzantine. Also, a fault can change the d and f values of a Byzantine process. Let the sequence $\langle x_1, x_2, x_3, x_4\rangle$ denote the set of states with respect to decision value of processes, i.e., $x_1 = d.g$, $x_2 = d.j$, $x_3 = d.k$, and $x_4 = d.l$. In this notation, an overlined (respectively, underlined) d-value shows that the corresponding process has finalized its decision (respectively, is Byzantine). Now consider the following scenarios:

- Starting from a state σ_0 in $\langle 1, \perp, \perp, 1\rangle$, where the general and process l agree on decision 1 and processes j and k are undecided, the program may reach the following sequence of states due to occurrence of faults (denoted $\dashrightarrow$) and execution of program actions (denoted $\rightarrow$): $\langle 1, \perp, \perp, 1\rangle \dashrightarrow \langle \underline{1}, \perp, \perp, 1\rangle \dashrightarrow \langle \underline{0}, \perp, \perp, 1\rangle \rightarrow \langle \underline{0}, 0, \perp, 1\rangle \rightarrow \langle \underline{0}, 0, 0, 1\rangle$. Let σ_1 be a state in $\langle \underline{0}, 0, 0, 1\rangle$, where the Byzantine general g and non-general processes j and k agree on decision 0, but process l has decided on 1. Now, consider the tasks for a synthesis algorithm in dealing with state σ_1. Note that no process can determine whether other processes have finalized their decision due to the issue of distribution. Thus, the synthesis algorithm rules out transitions that originate from σ_1 and j finalizes its decision, as it would violate safety (i.e., agreement). Likewise, it cannot allow k and l to finalize either. We call states such as σ_1 a *deadlock state*, since the program cannot proceed its execution. A synthesis algorithm can resolve this deadlock state by simply adding a *recovery* transition that changes the decision of l to 0 which results in reaching a legitimate state without violating safety. After adding such transitions, in the next iteration of the synthesis algorithm, we can allow j and k to finalize their decision after concluding that $\langle 0, 0, 0, \overline{1}\rangle$ (i.e., where l is not Byzantine and has finalized) is not reached.

- Now, consider the scenario where σ_0 reaches the following sequence of states: $\langle 1, \perp, \perp, 1\rangle \rightarrow \langle 1, \perp, \perp, \overline{1}\rangle \dashrightarrow \langle \underline{1}, \perp, \perp, \overline{1}\rangle \dashrightarrow \langle \underline{0}, \perp, \perp, \overline{1}\rangle \rightarrow \langle \underline{0}, 0, \perp, \overline{1}\rangle \rightarrow \langle \underline{0}, 0, 0, \overline{1}\rangle$. Let σ_2 be a state in $\langle \underline{0}, 0, 0, \overline{1}\rangle$, where non-general processes j and k agree with the

> Byzantine general on decision 0, but process l has finalized its decision on 1. Obviously, σ_2 is also a deadlock state. However, unlike σ_1 in the previous scenario, since process l has finalized its decision, we cannot resolve σ_2 by adding safe recovery. One approach to deal with such deadlock states is to simply eliminate them (i.e., making them unreachable). However, since we require that during elimination of a deadlock state, no new deadlock states must be created, a respective deadlock resolution algorithm involves many backtracking steps. In particular, in order to resolve σ_2, the algorithm needs to explore the reachability graph and remove the transition that allows a process to finalize its decision while there exist two undecided processes.

In Chapter 11, we observed that in order to automatically synthesize a fault-tolerant version of $\mathcal{BA}$ identical to the one by Lamport, Shostak, and Pease [LSP82], 96% of the total synthesis time is spent to resolve deadlock states.

With this motivation, in this chapter, we introduce a parallel BDD-based algorithm for resolving deadlock states in distributed programs that are subject to a set of faults. We specifically design our algorithm for multiprocessor architectures with shared memory (e.g., multi-core processors) due to their availability in virtually any organization. Intuitively, our algorithm partitions the transition relation of the given intolerant program across multiple threads where each thread works on a different processor core. The algorithm makes no assumptions about the structure of a given program (e.g., set of transitions, number of distributed processes, or its reachable states) in order to resolve deadlock states. Thus, we expect the algorithm to be generally applicable to a wide variety of distributed programs. Our parallel algorithm tends to require more memory than its sequential version. However, based on our experimental results, unlike model checking, BDD-based synthesis algorithms *run out of time* before they *run out of memory*. Hence, the increased space complexity is unlikely to be a bottleneck during synthesis.

We note that symbolic algorithms are known to be notoriously hard to parallelize due to the interdependence among data structures involved in such algorithms. As a matter of fact, while parallel implementations of symbolic model checkers are often successful in increasing available memory, the speedup gained from such techniques is limited. This is largely due to the irregular nature of the state-space generation task and the resulting high parallel overheads such as load imbalance and scheduling of small computations.

14.2 Parallel Symbolic Resolution of Deadlock States

In this section, we present our parallel BDD-based algorithm for resolving deadlock states reachable in the presence of faults in a distributed program. A major barrier in such parallelization is that BDD manipulation packages are not reentrant due to data structures shared across several BDDs (e.g., a hash table that stores all BDD nodes). There are two approaches to deal with this obstacle. The first approach is to modify a BDD package to make it reentrant. The second approach is to utilize multiple instances of the BDD package that do not share memory. With this approach, each thread works on its own copy of related BDDs. However, changes made by one thread would not be immediately available to other threads. Hence, threads may change the BDDs (e.g., the program being synthesized) inconsistently. Therefore, we need to *merge* the results and remove/manage the inconsistencies. In this work, we consider the second approach.

Algorithm sketch. Intuitively, our algorithm works as follows. During deadlock resolution, a *master* thread spawns several *worker* threads each running on a different processor core in parallel with an instance of its own BDD package. The instance of the BDD package assigned to each worker thread is initialized using BDDs for program transitions, invariant predicate, fault-span, and fault transitions. The master thread partitions the set of deadlock states and provides each worker thread with one such partition. Subsequently, worker threads start resolving their assigned set of deadlock states in parallel by either (1) adding *safe recovery*, or (2) *eliminating* the ones (i.e., making them unreachable) from where safe recovery is not possible. Upon completion, the master thread *merges* the results returned by each worker thread and resolves inconsistencies.

14.2.1 Parallel Addition of Safe Recovery

Given a program $\mathcal{P}$, faults F, fault-span $S_{\mathcal{P}}$, invariant predicate $Inv_{\mathcal{P}}$, safety specification $SPEC_{bt}$, and *partition predicates* $prt_1 \dots prt_n$, where $n \geq 1$ is the number of worker threads to be spawned, our goal is to synthesize a transition predicate $T_{\mathcal{P}'}$ such that $S_{\mathcal{P}}$ contains no deadlock states, i.e., $S_{\mathcal{P}} \wedge \neg Guard(T_{\mathcal{P}'}) = \mathit{false}$. Before we describe our parallel algorithm for resolving deadlock states through addition of recovery actions, notice that such a recovery mechanism should not violate the safety specification. Thus, we first identify the state

Algorithm 14.1 ResolveDeadlockStates

Input: program transition predicate $T_{\mathcal{P}}$, faults F, invariant $Inv_{\mathcal{P}}$, fault span $S_{\mathcal{P}}$, safety specification $SPEC_{bt}$, and partition predicates $prt_1..prt_n$, where n is the number of worker threads.

Output: program transition predicate $T_{\mathcal{P}'}$ and the predicate fte of states failed to eliminate.

1: Let rfo be the state predicate reachable by faults only from the invariant predicate;
2: Let ms be the state predicate from where faults alone can reach a state where $Guard(F \wedge SPEC_{bt})$ is true.
3: $mt := SPEC_{bt} \vee \langle ms \rangle'$;
4: $ds := S_{\mathcal{P}} \wedge \neg Guard(T_{\mathcal{P}})$;

{*// Resolving deadlock states by adding safe recovery*}
5: **for** $i := 1$ **to** n **do**
6: $rt_i :=$ SpawnThread $\rightsquigarrow$ AddRecovery$(ds \wedge prt_i,\ Inv_{\mathcal{P}},\ mt)$;
7: **end for**
8: ThreadJoin$(1..n)$;

9: $T_{\mathcal{P}} := T_{\mathcal{P}} \vee \bigvee_{i=1}^{n} rt_i$;
10: $vds, fte := false$;
11: $ds := S_{\mathcal{P}} \wedge \neg Guard(T_{\mathcal{P}})$;

{*// Eliminating deadlock states from where safe recovery is not possible*}
12: **for** $i := 1$ **to** n **do**
13: $rp_i, vds_i, fte_i :=$ SpawnThread $\rightsquigarrow$ Eliminate$(ds \wedge prt_i, T_{\mathcal{P}}, Inv_{\mathcal{P}}, F, S_{\mathcal{P}}, vds, rfo, fte)$;
14: **end for**
15: ThreadJoin$(1..n)$;

{*// Merging results from worker threads*}
16: $T_{\mathcal{P}'} := Group(\bigwedge_{i=1}^{n} rp_i)$;
17: $fte := \bigvee_{i=1}^{n} fte_i$;
18: $vds := \bigvee_{i=1}^{n} vds_i$;

19: $nds := ((S_{\mathcal{P}} \wedge \neg Inv_{\mathcal{P}}) \wedge \neg Guard(T_{\mathcal{P}'})) \wedge \neg((S_{\mathcal{P}} \wedge \neg Inv_{\mathcal{P}}) \wedge \neg Guard(T_{\mathcal{P}}))$;
20: $T_{\mathcal{P}'} := T_{\mathcal{P}'} \vee Group(T_{\mathcal{P}} \wedge nds)$;
21: $T_{\mathcal{P}'} := T_{\mathcal{P}'} \vee Group(T_{\mathcal{P}} \wedge \langle fte \wedge rfo \rangle')$;
22: **return** $T_{\mathcal{P}'},\ fte$;

predicate ms (Line 2 of ResolveDeadlockStates in Algorithm 14.1) from where faults alone can reach a state where $Guard(F \wedge SPEC_{bt})$ is true (i.e., faults alone can violate the safety). Now, let mt include the transitions in $SPEC_{bt}$ as well as transitions in $T_{\mathcal{P}}$ that end in ms. Observe that in order to ensure safety, $T_{\mathcal{P}'}$ (including its recovery actions) must be disjoint from mt.

After identifying the set ds of deadlock states in $S_{\mathcal{P}}$ (Line 4), we partition ds using the partition predicates such that $\bigvee_{i=1}^{n}(prt_i \wedge ds) = ds$. To efficiently partition deadlock states between threads, one needs to design a method such that (1) deadlock states are evenly distributed among worker threads, and (2) states considered by different threads for eliminating have a small overlap during backtracking. Regarding the first constraint, we can partition deadlock states based on values of some variable and evaluate the size of corresponding BDDs by the number of minterms that satisfy the corresponding formula. Regarding the second constraint, we expect that the overhead for such a split is as high as it requires dedicated analysis of program transitions. Hence, instead of satisfying this constraint, we add synchronization between threads. Thus, we design partition predicates based value of variables. For example, in the case of Byzantine agreement program with four worker threads, we let $prt_1 = (d.j = 0) \wedge (d.k = 0)$, $prt_2 = (d.j = 0) \wedge (d.k \neq 0)$, $prt_3 = (d.j \neq 0) \wedge (d.k = 0)$, and $prt_4 = (d.j \neq 0) \wedge (d.k \neq 0)$. Next, we assign each partition $prt_i \wedge ds$ of deadlock states to a worker thread to identify safe recovery paths from $prt_i \wedge ds$ to the invariant predicate in a layered fashion (Lines 5-8 in Algorithm ResolveDeadlockStates).

Each worker thread for adding recovery works as follows (cf. AddRecovery in Thread 14.2). Let the first layer, *lyr*, be the invariant predicate S (Line 1). We now construct the recovery transition predicate *rt* by (1) including transitions that originate from the given set of deadlock states *ds* and end in *lyr* (Line 3), and (2) excluding transitions that can lead the program to a state where safety may be violated (Line 4). We add the resulting recovery transition predicate to *rec* (Line 5). Now, for the next iteration, we let *lyr* be the state predicate from where one-step safe recovery is possible (Line 6). We continue adding recovery transition predicates until no such transition predicate is added. Notice that our strategy on adding recovery paths guarantees that no cycles are introduced to the fault-span. Hence, any computation that takes a recovery path reaches the invariant predicate in a finite number of steps.

Once all worker threads complete there job (Line 8 in Algorithm 14.1), the master thread adds all the recovery transitions returned by worker threads to the program's transition predicate (Line 9 in Algorithm ResolveDeadlockStates). At this point, the remaining deadlock states (Line 11) have to be made unreachable, as it is not possible to add safe recovery from them to the invariant predicate.

Thread 14.2 AddRecovery

Input: deadlock states ds, invariant $Inv_{\mathcal{P}}$, and transition predicate mt.
Output: recovery transition predicate rec.

$lyr,\ rec := Inv_{\mathcal{P}},\ false;$
repeat
  $rt := Group(ds \wedge \langle lyr \rangle');$
  $rt := rt \wedge \neg Group(rt \wedge mt);$
  $rec := rec \vee rt;$
  $lyr := Guard(ds \wedge rt)$
until $(lyr = false);$
return $rec;$

Thread 14.3 Eliminate

Input: deadlock states ds, program $T_{\mathcal{P}}$, invariant $Inv_{\mathcal{P}}$, fault transitions F, fault span $S_{\mathcal{P}}$, visited deadlock states vds, states predicate reachable by faults only rfo, predicate fte failed to eliminate.
Output: revised program transition predicate $T_{\mathcal{P}}$, visited deadlock states vds, predicate fte failed to eliminate.

wait$(mutex);$
    $ds := ds \ \wedge \ \neg vds;$
    $vds := vds \ \vee \ ds;$
signal $(mutex);$
if $(ds = false)$ **then**
  return $T_{\mathcal{P}};$
end if

$old := T_{\mathcal{P}};$
$tmp := (S_{\mathcal{P}} \wedge \neg Inv_{\mathcal{P}}) \wedge T_{\mathcal{P}} \wedge \langle ds \rangle';$
$T_{\mathcal{P}} := T_{\mathcal{P}} \wedge \neg Group(tmp);$
$fs := Guard(S_{\mathcal{P}} \wedge \neg Inv_{\mathcal{P}} \wedge F \wedge \langle ds \rangle') \ \wedge \ \neg rfo;$
$T_{\mathcal{P}}, vds, fte :=$ Eliminate$(fs, T_{\mathcal{P}}, Inv_{\mathcal{P}}, F, S_{\mathcal{P}}, vds, rfo, fte);$
$nds := Guard(S_{\mathcal{P}} \wedge \neg Inv_{\mathcal{P}} \wedge Group(tmp) \wedge \neg Guard(T_{\mathcal{P}}));$
$T_{\mathcal{P}} := T_{\mathcal{P}} \vee (Group(tmp) \wedge nds);$
$nds := nds \wedge Guard(tmp);$
$fte := fte \vee \neg\langle old \wedge \neg T_{\mathcal{P}} \wedge S_{\mathcal{P}} \wedge \langle ds \rangle' \rangle'';$
$T_{\mathcal{P}}, vds, fte :=$ Eliminate$(nds \wedge \neg Inv_{\mathcal{P}}, T_{\mathcal{P}}, Inv_{\mathcal{P}}, F, S_{\mathcal{P}}, vds, rfo, fte);$
return $T_{\mathcal{P}},\ vds,\ fte;$

Example. As mentioned in the introduction, one type of deadlock states in $\mathcal{BA}$ is of the form $\langle \underline{0}, 0, 0, 1 \rangle$, where the Byzantine general g and non-general processes j and k agree on decision 0, but process l has decided on 1. The algorithm ResolveDeadlockStates resolves such deadlock states and their symmetrical states by adding the following recovery actions to process l (and by symmetry to processes j and k) of $\mathcal{BA}$:

$$\mathcal{BA}3_l \; :: \; d.j = 0 \; \wedge \; d.k = 0 \; \wedge d.l = 1 \; \wedge \; f.l = 0 \qquad \longrightarrow \qquad d.l, f.l := 0, 0|1$$
$$\mathcal{BA}4_l \; :: \; d.j = 1 \; \wedge \; d.k = 1 \; \wedge d.l = 0 \; \wedge \; f.l = 0 \qquad \longrightarrow \qquad d.l, f.l := 1, 0|1$$

14.2.2 Parallel State Elimination

Let *ds* be a deadlock state predicate from where recovery to the invariant predicate cannot be added. Hence, in order for $\mathcal{P}'$ (the synthesized program) to satisfy the third condition of the synthesis problem, we need to ensure that *ds* is eliminated from the set of states that $\mathcal{P}'$ can reach in the presence of faults. Similar to addition of recovery paths, the Algorithm ResolveDeadlockStates launches one worker thread per each partition of *ds* for elimination (Lines 12-15).

The Thread Eliminate (cf. Thread 14.3) works as follows. We first keep track of *visited deadlock states* by all worker threads (Lines 1-4) so that no thread attempts to eliminate deadlock states that have already been considered for elimination. In particular, all threads synchronize on the predicate *vds* which contains visited deadlock states by all threads (Lines 1-4). Next, we remove all incoming transitions to *ds* (Lines 8-10). Then, since a program does not have control over the occurrence of faults, we eliminate states that can reach *ds* via a fault transition (Lines 11-12). Now, if removal of transitions in Line 10 causes some state predicate *nds* to become a deadlock state predicate (Line 13) then we add the transitions (and the corresponding group) that begin from *nds* (Lines 15-17) to $T_{\mathcal{P}}$ and instead, we eliminate *nds*[1]. We keep repeating this procedure recursively until there does not exist a state to eliminate.

Once all worker threads complete their job (Line 15 in Algorithm 14.1), the master thread merges all the results by collecting transitions that all worker threads agree on (Line 16). Although the above algorithm is a sound building block for a sequential algorithm, it may create inconsistencies when multiple instances of it run in parallel.

Handling Inconsistencies. Let σ_1 and σ_2 be two states that are considered for elimination and (σ_0, σ_1) and (σ_0, σ_2) be two transitions for some σ_0. A sequential algorithm

[1]Let $T_{\mathcal{P}}$ be a transition predicate. $\langle T_{\mathcal{P}} \rangle''$ denotes the state predicate obtained by first abstracting unprimed variables in $T_{\mathcal{P}}$ and then replacing all primed variables of $T_{\mathcal{P}}$ by their corresponding unprimed variables.

that applies Eliminate, removes transitions (σ_0, σ_1) and (σ_0, σ_2) which causes σ_0 to be a new deadlock state (cf. Algorithm 14.1). Hence, it puts (σ_0, σ_1) and (σ_0, σ_2) (and corresponding group predicates) back into the program being synthesized and invokes Eliminate on state σ_0. However, when multiple worker threads, say th_1 and th_2, run concurrently, there are three possible scenarios that cause inconsistencies, described next.

Case 1. Consider the case where deadlock states σ_1 and σ_2 are in different partitions. Hence, th_1 invokes Eliminate on σ_1 which in turn removes (σ_0, σ_1), and, th_2 invokes Eliminate on σ_2 which removes (σ_0, σ_2) (cf. Figure 14.1.b). Thus, neither thread invokes Eliminate on σ_0, since they do not identify σ_0 as a deadlock state. Subsequently, when the master thread merges the results returned by th_1 and th_2 (i.e., Line 16 in Algorithm 14.1), σ_0 becomes a new deadlock state which has to be eliminated while the group predicates of transitions (σ_0, σ_1) and (σ_0, σ_2) have been removed unnecessarily. In order to resolve this case, we replace all outgoing transitions that start from σ_0 and mark σ_0 as a state that has to be eliminated in subsequent iterations (Lines 19-20).

Case 2. Due to backtracking behavior of Eliminate, it is possible that th_1 and th_2 consider common states for elimination. In particular, if th_1 considers σ_1 and th_2 considers both σ_1 and σ_2 for elimination (cf. Figure 14.1.b), after merging the results, no new deadlock states are introduced. However, (σ_0, σ_1) would be removed unnecessarily. In order to resolve this case, we collect all the states that worker threads failed to eliminate (i.e., state predicate *fte* in Line 17 in Algorithm 14.1) and replace all incoming transitions into those states (Line 21).

Case 3. It is also possible that th_1 considers σ_1 and th_2 considers neither σ_1 nor σ_2 (cf. Figure 14.1.c). This case occurs when th_2 stops backtracking at a level higher than σ_1 and σ_2 in the reachability graph due to facing either Case 1 or Case 2. Thus, when the master thread

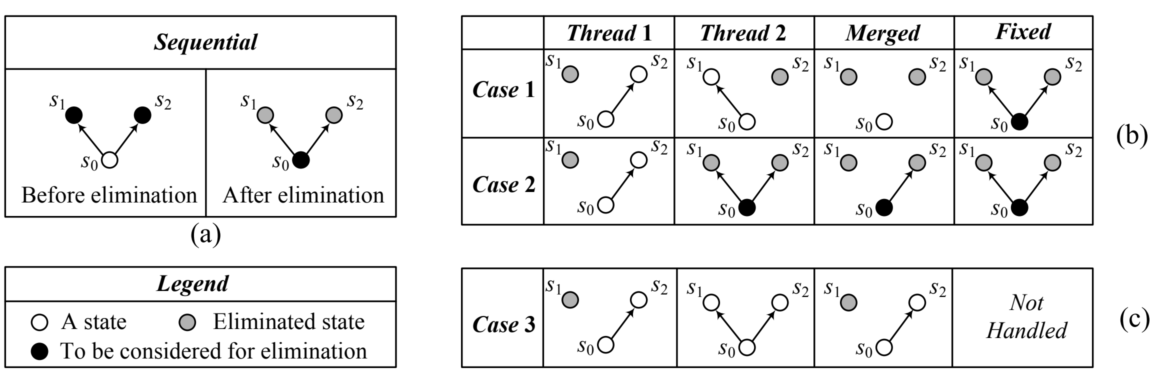


Figure 14.1: Inconsistencies raised by concurrency.

merges the results returned by the worker threads, no new deadlock state is introduced, but (σ_0, σ_1) is removed unnecessarily. While identifying this case given the structures in Figure 14.1.c is not straightforward, one approach to resolve this inconsistency is to force all worker threads to synchronize at each backtracking step. Since such synchronization seems to decline the performance of the parallel algorithm, we choose not to handle this case. Notice that removal of (σ_0, σ_1) does not result in synthesizing an incorrect program. However, the program synthesized using the parallel algorithm may have less transitions than the program synthesized by the sequential algorithm. We note that this case is not due to our algorithm strategy, but an artifact of breadth-first-search nature of BDD-based reachability analysis. In fact, any random state space search strategy may as well exhibit this case.

Example. As mentioned in the introduction, another type of deadlock states in $\mathcal{BA}$ is of the form $\langle \underline{0}, 0, 0, \overline{1} \rangle$, where non-general processes j and k agree with the Byzantine general on decision 0, but process l has finalized its decision on 1. Since process l has finalized its decision, we cannot resolve such deadlock states by adding safe recovery. Thus, the algorithm ResolveDeadlockStates has to eliminate states in $\langle \underline{0}, 0, 0, \overline{1} \rangle$. More specifically, the Thread Eliminate backtracks through the reachability graph until it removes the transition $\langle 1, \perp, \perp, 1 \rangle \rightarrow \langle 1, \perp, \perp, \overline{1} \rangle$. This removal creates no new deadlock state and, hence, Eliminate terminates successfully. Precisely, our algorithm revises action $\mathcal{BA}2_l$, so that no computation of $\mathcal{BA}$ in the presence of faults reaches a deadlock state as follows:

$$\mathcal{BA}2_l \;::\; (d.l \neq \perp) \;\wedge\; (f.l = \mathit{false}) \;\wedge\; (d.j \neq \perp \;\vee\; d.k \neq \perp) \qquad \longrightarrow \qquad f.l := \mathit{true}$$

We note that in the context of of $\mathcal{BA}$, inconsistency of type Case 3 does not occur. However, Cases 1 and 2 do occur, but our algorithm fixes them. In fact, the output of our synthesis algorithm is identical to the solution proposed by Lamport, Shostak, and Pease [LSP82].

Part V

Literature Survey and Conclusion

Chapter 15

Related Work

In this chapter, we illustrate how the contributions of this dissertation differ from existing approaches in program synthesis. Before we discuss the related work in the context of program revision and synthesis, we note that like many other words in computer science, the term "synthesis" is used to mean different things in different contexts. These contexts range from code generation from (formal or informal) specification to transformation, be it fully automated, semi-automated, or fully manual. In this dissertation, we used the term "synthesis" in the context of program "revision", where we focus on fully automated transformation of one *abstract program* into another *abstract program* that meets additional requirements (e.g., safety, timing constraints, fault-tolerance). Thus, in this chapter, we focus on related work that (1) are fully automated, (2) their output is correct-by-construction, and (3) deal with programs represented by state-transition systems. Thus, the following lines of research are orthogonal to our approach in this dissertation:

1. **Platform and architecture dependent techniques.** There exists an extensive line of research in the systems area on fault-tolerant real-time and distributed computing. For instance, in the literature of real-time computing, most approaches address fault-tolerance in the context of scheduling theory (e.g., [PM98, AM99a, LMM00, AAMM00, MMG03, AMM00]). In fault-tolerant real-time scheduling, the objective is to find the optimal schedule of a set of tasks on a set of processors dynamically, such that the largest possible number of tasks meet their deadlines. Since time complexity is a critical issue in dynamic scheduling, most of the proposed algorithms are best-effort heuristics designed for specific platforms and special types of faults (e.g.,

transient, fail-stop, Byzantine, etc). We note that the output of our algorithms is an abstract program that may be further refined into a deterministic program in a high-level language for a specific platform and architecture.

2. **Logic-based program synthesis.** Deductive program synthesis methods (e.g., [MW80, MW92, YqRzH85, CK89]) rely on a theorem-proving approach. This approach combines techniques of unification, mathematical induction, and transformation rules within a single deductive system. However, since eductive program synthesis methods do not observe a program as a state-transition system, they are not quite relevant to the scope of this dissertation. Although it is unclear how one can use theorem proving techniques in state-transition graph-based synthesis or vice versa, it is certainly an interesting question for further investigation.

3. **Manual techniques.** Dijkstra [Dij90], Chandy and Misra [CM88], and Gries [Gri81] propose a wide variety of techniques for stepwise development of correct programs. When using their methodology, a programmer begins with a specification of the problem written in an assertion language such as predicate logic and incrementally refines the specification into code. Such a refinement technique is intended for human use, as it may require intuition of the programmer during the application process. In Section 15.4, we discuss how (possibly manual) pre-synthesized program components can be used to for adding fault-tolerance to a program.

The rest of this chapter is organized as follows. In Section 15.1, we present the related work on synthesis of programs in closed systems. Then, in Section 15.2, we present the related work on program synthesis in open systems. Section 15.3 is dedicated to describe state-of-the-art synthesis tools. Finally, we discuss the related work on the theory of detectors and correctors in Section 15.4.

15.1 Program Synthesis in Closed Systems

In this section, we present the related work on automated synthesis of closed systems. Specifically, in Subsection 15.1.1, we discuss the line of research on comprehensive synthesis of closed systems. Then, in Subsection 15.1.2, we present comparison and contrast between our work and the related work in the area of program correction and repair.

15.1.1 Comprehensive Synthesis

Most existing comprehensive synthesis methods for closed systems focus on deriving the synchronization skeleton of a program from its specification in terms of a temporal logic expressions [EC82, MW84, AAE04, Att99, AE01]. The synchronization skeleton of a program is an *abstract structure* of the code of the program implementing inter-process synchronization. Although such synthesis methods differ in the type of input specification language and the program model that they synthesize, they are all based on a proof that the specification is satisfiable. This makes it difficult to provide reuse during synthesizing programs. In other words, any change in the specification requires the synthesis to be restarted from scratch. To the contrary, in our approach, since the input to our algorithms is an existing program, our methods have the potential to reuse the structure of the program incrementally add new properties.

The seminal work in this area is due to Emerson and Clarke [EC82], where they propose a tableau-based method for deriving a finite state model from a CTL formula. Specifically, their synthesis method builds a tableau, which contains all potential models. Thus, if a formula f is satisfiable, then a model of f exists in the tableau. And, if there indeed exists a model, then their method extracts a synchronization skeleton of the model from the tableau. Briefly, a tableau for a CTL formula is a finite directed AND/OR graph. This graph is built systematically in two steps: (1) setting the formula as the root of the tableau, and (2) continuing the reduction of unreduced AND-nodes and OR-nodes, while there exists an unreduced node (i.e., a node without successor).

Manna and Wolper [MW84] present a similar method for synthesizing distributed communicating processes from a formula in Propositional Linear Temporal Logic. The output model of [MW84] stipulates that all the inter-process communications (through message passing) take place through a synchronizer process.

Attie and Emerson study the problem of synthesizing concurrent programs from CTL specifications in a line of papers. Specifically, in [AE98], they address the state explosion problem using the behavioral similarity of processes of a concurrent program. They argue that in a set of concurrent processes, similarity in behavior often exists. For instance, in a mutual exclusion problem, a set of process are competing to acquire a critical section. Attie and Emerson also address the problem of synthesizing distributed processes using the method in [EC82] and by incorporating atomic read/write actions [AE01].

Algorithms for comprehensive redesign of timed automata [AD94] from real-time temporal logic Mitl formulae was first introduced in [AFH96]. More recently, in [MNP06], the authors present much simpler algorithms for constructing timed automata from Mitl formulae than the ones in [AFH96].

15.1.2 Program Repair and Correction

There has been an extensive line of research in the area of correction of combinational hardware circuits. In this line of research, the notion of *fault* does not have exactly the same meaning as what we defined in Chapter 7. In particular, a fault is observed as: any gate can be replaced by an arbitrary function. In [MCB89, LTL90, PYC94], the authors present formal methods for fault localization and correction based on Boolean equations. Similar to most formal methods, in these papers, the correct behavior of circuits is given as a specification. We note that circuits consider in the aforementioned papers are sequential.

Motivated by the same concept, Jobstmann, Griesmayer, and Bloem [JGB05] formulate the problem of program repair as a two player Büchi game. In their framework, the specification is given in Linear Temporal Logic, and state the correction problem as a game, in which the protagonist selects a faulty component and suggests alternative behavior. The objective of their work is very similar to our goal in Part II of this dissertation. In particular, the authors have independently shown that deciding whether a memoryless strategies exists is NP-complete. Existence of a memoryless strategy for a Büchi game can be precisely translated to adding two eventually properties to an untimed centralized program (see Theorem 4.2.1).

Buccafurri et al. [BEGL99] study the correction problem for Ctl as an abductive reasoning problem. The authors present a method based on invoking a model checker once for every possible correction to see if it is successful. Our approach, on the other hand, needs to consider the problem only once, considering all possible corrections at the same time, and is likely to be more efficient. In fact, in our approach, we do not need to call a model checker at all.

Finally, Janjua and Mycroft [JM06] describe how to automatically insert synchronization statements in a multi-threaded program in order to prevent bugs due to an unfortunate scheduling. In [SLTB$^+$06], the authors propose a C-like language, in which programs can be modeled. A synthesizer then completes the model so that it adheres to a specification.

15.2 Program Synthesis in Open Systems

In this section, we present the related work on automated synthesis of open systems. Specifically, in Subsection 15.2.1, we discuss the line of research on incremental synthesis of fault-tolerant programs. Then, in Subsections 15.2.2 and 15.2.2, we present comparison and contrast between our work and the related work in the area of discrete controller synthesis and game theory, respectively.

15.2.1 Automated Synthesis of Fault-Tolerance

The problem of synthesizing *untimed* fault-tolerant programs has been studied in the literature from different perspectives. The seminal work in this area is due to Attie, Arora, and Emerson, where they study the problem of synthesizing fault-tolerant concurrent untimed programs from temporal logic specifications expressed in CTL formulae [AAE04]. Their approach is essentially based on Emerson and Clarke's tableau-based method [EC82]. The input to the their algorithm is (1) a specification represented by a CTL formula, (2) fault actions and fault specification, which is a set of auxiliary atomic propositions, (3) the coupling specification, which is also a CTL formula, and (4) a level of fault-tolerance. The algorithm, first, generates a finite model of the program using the tableau-based method in [EC82]. Then, they apply the fault actions to every state of the generated model in order to generate new states, i.e., states that the program can reach in the presence of faults. Then, they generate recovery transitions that can reach the program invariant. At the end, they apply a set of modified deletion rules to prune the augmented global state-transition diagram in order to extract the individual processes. Similar to comprehensive synthesis approaches, this approach cannot start from a given intolerant program. Thus, the lack of reuse is the key difference between our approach and the method in [AAE04].

The algorithms for automatic addition of fault-tolerance [KA00, KAC01, KE02, KE03, KE04] add fault-tolerance concerns to existing untimed programs in the presence of faults, and guarantee the addition of no new behaviors to the original program in the absence of faults. In the seminal work in this area [KA00], the authors introduce synthesis methods for automated addition of fault-tolerance to untimed centralized and distributed programs.

In particular, they introduce polynomial-time sound and complete algorithms for adding all levels of fault-tolerance (failsafe, nonmasking, and masking) to centralized programs. The input to these algorithms is a fault-intolerant centralized program, safety specification, and a set of fault transitions. The algorithms generate a fault-tolerant program along with an invariant predicate. The authors also 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. We note that Theorem 5.2.1 in this dissertation is a generalization of the NP-completeness result in [KA00]. This is because in Theorem 5.2.1, we showed that the problem of adding a progress property (in other words, adding recovery in [KA00]) to a distributed program is NP-complete even in the absence of faults.

In order to cope with the exponential complexity identified in [KA00], Kulkarni, Arora, and Chippada introduce a set of heuristics to solve the problem of synthesizing distributed masking programs in polynomial-time. Our heuristics in Chapter 11 are based on the ones in [KAC01] with the obvious difference that our heuristics are symbolic. Unfortunately, there is no published data on performance analysis of the heuristics in [KAC01].

In [KE02, KE05b], Kulkarni and Ebnenasir show that the problem of adding failsafe fault-tolerance to distributed programs is also NP-complete in the size of the input program's state space. They also identify a class of specifications, monotonic specifications, and a class of programs, monotonic programs, for which the synthesis of failsafe fault-tolerance can be done in polynomial-time (in program state space). Moreover, they prove that if only one of the monotonicity conditions is satisfied, the synthesis of failsafe fault-tolerance is still NP-complete. Intuitively, the notion of of monotonicity captures the case where a program $\mathcal{P}$ can safely assume that a variables x is false and, even if x were true when $\mathcal{P}$ executes, the corresponding transition would not violate safety. We emphasize that Theorem 5.1.1 in this dissertation is a generalization of the NP-completeness result in [KE02, KE05b]. This is because in Theorem 5.1.1, we showed that the problem of adding only one safety property (in other words, removing unsafe transition in [KE02, KE05b]) to a distributed program is NP-complete even in the absence of faults.

Kulkarni and Ebnenasir [KE03] also present algorithmic solutions for enhancing the level of fault-tolerance of programs from nonmasking to masking. The problem of enhancing fault-tolerance from nonmasking to masking requires that safety be added and recovery be preserved. The authors present a sound and complete algorithm for centralized programs and a sound algorithm for distributed programs. Obviously, the algorithms in this work are useful for the case where the input program at hand is already nonmasking.

In [KE04], Kulkarni and Ebnenasir address the problem of automated synthesis of *untimed multitolerant* programs, i.e., programs that tolerate multiple classes of faults and provide a (possibly) different level of fault-tolerance to each class. They show that if one needs to add failsafe (respectively, nonmasking) fault-tolerance with respect to one class of faults and masking fault-tolerance with respect to another class of faults, then such addition can be done in polynomial-time in the size of state space of the fault-intolerant program. The novelty of their algorithm is, it adds fault-tolerance in a stepwise fashion. They, however, show that if one needs to add failsafe fault-tolerance with respect to one class of faults and nonmasking fault-tolerance with respect to another class of faults, then the problem is NP-complete.

Jhumka and Gärtner [GJ04] consider the problem of automated synthesis of failsafe programs, where the safety specification is not fusion closed. In other words, the (timing independent) safety specification (i.e., $SPEC_{bt}$) cannot be represented by a set of bad transitions; rather, by a set of bad prefixes. In general, fusion closure of specifications can be achieved by adding history variables. However, as Jhumka and Gärtner argue, the addition of history variables causes an exponential growth of the state space of the program, causing addition of fault tolerance to be expensive. To address this problem, the authors present a method which can be used to add history information to a program in a way that significantly reduces the number of states added due to incorporating history variables.

Kulkarni and Ebnenasir [KE05b] also investigate the effect of non-fusion closed safety specifications on the complexity of adding masking fault-tolerance to untimed centralized programs. As mentioned earlier, it is known that adding masking fault-tolerance to untimed centralized programs can be achieved in polynomial-time [KA00]. However, the authors in [KE05b] show that for the case where one represents the safety specification by a set of *bad pairs* of transitions, the problem of adding masking fault-tolerance to untimed centralized programs is NP-complete. We note that this NP-completeness result is our most fundamental reason for representing (timing independent) safety specification (see Definition 7.1.6) by a set of bad transitions in our framework.

Ebnenasir [Ebn07] develops a method for dividing the revision problem that scales up. In an application of this approach for safety properties, Ebnenasir develops an algorithm that statically analyzes (and possibly revises) program instructions on separate machines in a parallel/distributed platform. Based on this method, the author implements a distributed framework that exploits the computational resources of wide area networks for program revisions. Using this approach, it is possible to synthesize failsafe Byzantine agreement with 40 processes on a cluster of three machines in 353 seconds. Using our BDD-based approach, the same program can be synthesized in 22 seconds.

The problem of online fault *detection* in timed automata is studied in [Tri02]. In this work, Tripakis proposes a polynomial-space online algorithm for designing a diagnoser that detects faults in behaviors of a given timed automaton after they occur. It is assumed that (1) the given system is in synchronous model, and (2) faults and failures are identical events. Thus, this model does not capture situations where the occurrence of faults (although undesirable) is common and expected, but may *lead* a system to failures. Bouyer, Chevalier, and D'Souza [BCD05] address the same problem where the diagnoser is realizable as a deterministic timed automaton or an event record automaton.

15.2.2 Controller Synthesis

Synthesis of discrete-event systems has mostly been studied in the context of controller synthesis and game theory. In this subsection, we compare and contrast our approach to controller synthesis and game theory. The seminal work in the area of controller synthesis is due Ramadge and Wonham [RW89]. The discrete controller synthesis (DCS) problem is as follows: starting from two languages $\mathcal{U}$ and $\mathcal{D}$, identify a third language $\mathcal{C}$ such that $\mathcal{U} \cap \mathcal{C} \subseteq \mathcal{D}$. In DCS terminology, the three languages $\mathcal{U}$, $\mathcal{D}$, and $\mathcal{C}$ are called the *plant*, the *desired system*, and the *controller*, respectively. $\mathcal{U} \cap \mathcal{C}$ is called the *controlled system*. Finally, the set $\mathcal{A}$ of alphabets is represents events that can occur. Obviously, the languages $\mathcal{U}$ and $\mathcal{D}$ may represent the set of computations of a given program and a safety and/or reachability specification. Moreover, $\mathcal{C}$ identifies the computations that do not violate $\mathcal{D}$ in the presence of uncontrollable transitions.

One can notice that our work in this dissertation is in spirit close to DCS. Specifically, an input program and fault transitions may be modeled as controllable and uncontrollable actions. In fact, in both problems, the objective is to *restrict* the program actions at each state through synthesizing a controller such that the behavior of the entire system is always *desirable* according to safety and reachability conditions, in the presence of an adversary. As mentioned in Section 7.3, notice that the conditions $C1$ and $C2$ precisely express this notion of restriction. Furthermore, the conjunction of all conditions expresses the notion of *language inclusion*, where the synthesized program is supposed to exhibit a subset of behaviors of the input intolerant program. Having said that, our work differs from synthesizing discrete-event controllers in that:

1. The computation model for synthesizing controllers is based on prioritized synchronization, whereas ours is based on interleaving.

2. The complexity of synthesizing a fault-tolerant design in the context of the formulation presented in Problem Statement 7.3.2 for untimed centralized programs is polynomial-time whereas the complexity of synthesizing controllers is NP-hard [GW00].

3. Our algorithms are concerned with properties typically used in specifying real-time and distributed fault-tolerance requirements and, hence, they synthesize programs more efficiently.

4. In controller synthesis, the notion of addition of *recovery* computations does not exist, which is a crucial concept in fault-tolerant systems. To the best of our knowledge, this difference exists between SYCRAFT and virtually all tools that implement controller synthesis as well.

5. Finally, we model *distribution* by specifying read/write restrictions, whereas the issue of decentralized plants is modeled through *partial observability* [LW90, RW92].

In the context of real-time systems, Asarin, Maler, and Pnueli [AMP95] introduce a symbolic method to synthesize timed controllers. At the semantic level their approach synthesize a controller is synthesized by finding a winning strategy for either safety or reachability games (but not both) defined by traditional finite state automata or by timed-automata. The complexity gap between our approach and timed controller synthesis approaches is more visible. For example, the synthesis problems presented in [FLM02, AMPS98, AM99b, dAFH$^+$03] are EXPTIME-complete. Moreover, deciding the existence of a controller in [DM02, BDMP03] is 2EXPTIME-complete. By contrast, as we showed in Chapter 8, the complexity of our algorithms is significantly less.

15.2.3 Game Theory

Game-theoretic approaches for the synthesis of controllers and reactive programs [PR89a] are generally based on the model of two-player games [Tho95]. In such games a program makes moves in response to the moves of its environment. The program and its environment interact through a set of interface variables and, hence, the environment can only update the interface variables. In our model, however, faults can perturb all program variables. Moreover, in a two-player game model, players take turns and the set of states from where the first player can make a move is disjoint from the set of states from where the second player can move [WHT03]. To the contrary, in our work, fault-tolerance should be provided against faults that can execute from any state.

Game theoretic methods are based on the theory of tree automata [Tho90]. Such an automaton represents the specification of a system. A synthesis algorithm checks the non-emptiness of the automaton, i.e., whether there exists a tree acceptable by the tree automaton. If the tree automaton is indeed nonempty, then the specification is called *realizable* and there exists a model of the synthesized program.

Pnueli and Rosner address the problem of synthesizing synchronous open reactive modules in [PR89a]. They generalize their method in [PR89b], by proposing a technique for synthesizing asynchronous reactive modules. In particular, they investigate the problem of synthesizing an asynchronous reactive module that include only one process and interacts with a non-deterministic environment through a Boolean variable x and a Boolean output variable y. In order to define the realizability problem, Pnueli and Rosner define a program as a function from the set of finite sequences of input values to a sequence of output values. Given a domain D for input and output values, a program $\mathcal{P}$ is a function $f : D^+ \rightarrow D$ (D^+ represents the set of sequences of input values), where the variables x and y range over D. Thus, for every step i of execution, the program generates $f(a_0, a_1 \cdots a_i)$ as the value of y, where a_k is the observable value of x at step k, $0 \leq k \leq i$. Hence, each asynchronous behavior of the system consists of an infinite sequence of ordered pairs (a_i, b_i), where $a_i \in D$ is the input value of x and $b_i \in D$ is the output value of y at step i. Also, Pnueli and Rosner incorporate tow other infinite sequences that represent the scheduling state of the program. Finally, they define the realizability problem as follows: given an LTL specification $\phi(x, y)$, does there exist a program $\mathcal{P}$ that satisfies $\phi(x, y)$. They show that the necessary and sufficient condition for realizability of a specification is its validity, and not just satisfiability.

While symbolic model checking has been studied extensively (e.g., [BCM+92, McM93, HNSY94]), little work has been done on symbolic synthesis and especially on performance analysis. Recently, Wallmeier, Hütten, and Thomas [WHT03] introduce an algorithm for synthesizing finite state controllers by solving infinite games over finite state spaces. They model the winning constraint by safety conditions and a set of request-response properties as liveness conditions. They transform this game into a Büchi game which inevitably involves an exponential blow-up. The approach in [WHT03] does not address the issue of distribution or time. Moreover, the reported maximum number of variables in their experiments is 23, which is far less than the number of variables that we have handled using our symbolic algorithms.

Finally, we emphasize that similar to discrete controller synthesis, game theoretic approaches do not address the issue of addition of recovery. Also, in game theory, the notion of distribution is modeled by partial observability.

15.3 Synthesis Tools

The most relevant synthesis tool to SYCRAFT is FTSyn [EKAar]. This tool implements the synthesis heuristics introduced by Kulkarni, Arora, and Chippada [KAC01] in an enumerative fashion. The input language of FTSYn is similar to SYCRAFT; it accepts a set of processes along with read/write restrictions, safety specification, and an invariant. However, the grammar of input language of FTSyn is not as rich as SYCRAFT. For instance, FTSyn does not allow quantifiers and range on symmetric processes. It is also not possible to associate each process with a set of faults and prohibited transitions. Given an input program, the output of FTSyn is a masking fault-tolerant distributed program. Also, the output of SYCRAFT is more optimized than FTSyn in the sense that SYCRAFT simplifies Boolean minterms better than FTSyn. Unfortunately, at the time of writing this dissertation, there is no published data on performance of FTSyn. However, our experiments show a considerable difference between performance of FTSyn and SYCRAFT. For example, in case of Byzantine agreement for five processes, it take FTSyn about 15 minutes to generate an output, whereas SYCRAFT can generate the same output within less than a second.

Furthermore, FTSyn fails to add fault-tolerance to more than six non-general processes, whereas SYCRAFT has successfully been used to synthesize up to 50 non-general processes.

The tool LiLY [JB] synthesizes a functionally correct design from a formal specification expressed in LTL based on the new advances on optimizations for LTL synthesis [JB06]. In particular, LiLy is a tool for comprehensive synthesis from LTL specification written in Perl. The output of LiLY is a state machine represented as a Verilog[1] module or as a directed graph in DOT format. Anzu [JGWB07] is another tool from the same research group. The tool synthesizes Verilog designs from specifications written in LTL. Unlike LiLY, Anzu implements the Reactive-1 algorithm due to Pnueli, Piterman, and Sa'ar's [PPS06], which can handle a huge subset of LTL.

As mentioned earlier, Our synthesis problem is in sprit close to controller synthesis and game theory problems. However, there exist several distinctions in theories and, hence, the corresponding tools. In particular, in controller synthesis and game theory, the notion of addition of *recovery* computations does not exist, which is a crucial concept in fault-tolerant systems. To the best of our knowledge, this difference exists in virtually all tools that implement controller synthesis or game theoretic algorithms. Moreover, we model *distribution* by specifying read/write restrictions, whereas related tools either do not address the notion of distribution or their modeling is in a higher level of abstraction (e.g., the SMT-based method in [FS07]). Below, we present some of such tools. Some tools (e.g., TICC and Supremia) model the issue of distribution using synchronization on input actions.

SIGALI [BBG00] supports verification of reactive systems and synthesizing discrete controllers. In SIGALI, a system is represented by a labeled transition system, while the control of the system is performed by restricting the controllable input values to values suitable for the control goal. This restriction is obtained by incorporating new algebraic equations into the initial system. Similar to most controller synthesis approaches, in SIGALI the control objectives are reachability and persistence (e.g., a set of safe states). As mentioned earlier, in controller synthesis, some of the crucial features of fault-tolerance (e.g., safe recovery after a fault occurs) are not addressed. This is the case in SIGALI as well. Moreover, it is not possible to model distribution in SIGALI.

Supremica [AFFM06, AFFV03] is also a tool for verification, synthesis, and simulation

[1]Verilog is a hardware description language (HDL) used to model electronic systems. For more information, visit `http://www.verilog.com/`.

of discrete event systems. Similar to SIGALI, the input model in Supremica is a finite state automaton, where the transitions have an associated event together with a guard condition and an action function that updates the system variables. Supremica exploits modularity in order to address the state explosion problem, a feature that SYCRAFT currently lacks. Similar to SYCRAFT, implementations of Supremica is BDD-based. Another difference between SYCRAFT and Supremica is that input programs to SYCRAFT are modeled in an asynchronous setting, whereas in Supremica different transitions of the given automaton synchronize on input actions. Thus, comparison of experimental results is irrelevant.

TCT [FW06] is also a tool for supervisory control synthesis of untimed finite state systems. It incorporates BDDs in implementation of its core algorithms. Also, TCT exploits system hierarchical structure in order to remedy the state explosion problem. The latest version of TCT can compute systems that would need about 10^{20} states if computed extensionally in automata. We note that although TCT and SYCRAFT implement different models, as far as synchronization is concerned, SYCRAFT is able to synthesize programs with 10^{50} reachable states an beyond.

In [dAFL06], de Alfaro, Faella, and Legay introduce the tool TICC for game-based modeling of software and distributed systems. The core of TICC is based on the theory of interface automata [dAH01]. In TICC, components are modeled via both discrete variables and actions (to describe synchronization). The implementation of TICC is also BDD-based. In particular, TICC can synthesize input assumptions of *some* environment in which a distributed program can be executed without reaching a deadlock due to unanticipated moves by the environment. Similar to Supremica and unlike SYCRAFT, TICC models distribution concerns as *modules* that synchronize on shared actions.

15.4 Component-Based Analysis of Fault-Tolerant Programs

In Chapter 10, we extended *the theory of fault-tolerance components* [AK98b] to the context of real-time programs. The theory in [AK98b] essentially separates fault-tolerance and functionality concerns of untimed systems. More specifically, the theory identifies two types of fault-tolerance components, namely *detectors* and *correctors*. These components are based on the principle of detecting a *state predicate* to ensure that program actions would be safe and correcting a *state predicate* to ensure that the program eventually reaches a legitimate state. We emphasize that since these components do not rely on *detecting faults* or *correcting faults*, they can be applied in cases where faults are not detectable (e.g., Byzantine faults).

Our work in Chapter 10 differs from the work on failure detectors (e.g., the line of research pioneered by Chandra and Toueg) in that the predicates being detected in [CT96, CHT96] are of the type "process j has failed". To the contrary, the predicates in our work are arbitrary state predicates. Moreover, in [CT96, CHT96], the authors have considered detectors that are not perfect; similar detectors can also be constructed from the components in this paper. However, this issue is important in the context of a *design* methodology and is discussed in [BK06b, KE04, AK98a]. Since the discussion about *imperfect* detectors is not needed in our context, this issue is outside the scope of this paper. Likewise, issues such as *atomicity* of the fault-tolerance components are important for design; in the context of analysis, the components *contained* in a fault-tolerant program, by definition, would satisfy any atomicity restrictions imposed on that fault-tolerant program.

Although detectors and correctors have been found to be useful in the *design* of fault-tolerant programs [2], their significance in analysis has not been evaluated except in empirical case studies. In these studies [KRS99, GJ04], decomposition of a fault-tolerant program into its components has been found valuable in formal verification of the program. Thus, we expect that our affirmative answer to existence of the components in Chapter 10 would significantly assist in analysis of real-time embedded fault-tolerant programs.

The theory of detectors and correctors [AK98b] was extended in [JGFS02] for safety-critical systems. In [TG99], the authors have used a similar approach for proving convergence of systems to legitimate states. The theory has also been used in design of several multi-tolerant examples [KE04, GH00] where tolerance to different types of faults is provided and the level of fault-tolerance varies depending upon the severity of faults. In the context of automation of addition of fault-tolerance, the theory has been exploited in [BK08a, BK06b, KE04, GJ04, JHS02]. In the context of verification, simplified versions of this theory are applied in verification of time-triggered architectures [Rus02]. It has also been used in software verification through separation of concerns [KRS99, JHA07].

[2] The components have been shown to suffice in the design of a large class of fault-tolerant programs [AK98b] including programs designed using replication, state machine approach, and, checkpointing and recovery.

In [KE05a], the authors consider the problem of incorporating pre-synthesized detector and corrector components for synthesizing masking fault-tolerant distributed programs. The motivation of their work is based on the facts that the problem of synthesizing distributed masking programs is NP-complete and heuristics introduced in [KAC01] may fail to synthesize a program. Similar to our approach in this dissertation, their synthesis method reuses the given fault-intolerant program. It also ensures interference freedom among the given pre-synthesized components and the intolerant program, i.e., the execution of one of them does not violate the (safety or liveness) specification of another one. Obviously, such a synthesis method is semi-automated as the designer has to somehow provide their synthesis method with the right pre-synthesized components.

Based on the detector-corrector theory Ebnenasir and Cheng [EC07] introduce an object analysis pattern, called the detector pattern, that provides a reusable strategy for capturing the requirements of failsafe fault-tolerance in an existing conceptual model. They also present a method that uses the detector pattern for creating a behavioral model of a failsafe fault-tolerant system in UML. Their method also generates and model checks formal models of UML state diagrams of the fault-tolerant system. In addition it visualizes the model checking results in terms of the UML diagrams. Ebnenasir and Cheng [EC06] also introduce the notion of correctors pattern in order to model and analyze error recovery in fault-tolerant programs.

Chapter 16

Conclusion and Future Work

In this dissertation, we focused on the problem of *automated revision of distributed and real-time programs* in the context of *open* and *closed* systems. In both cases, we concentrated on properties that are typically used in specifying distributed and real-time systems (e.g., UNITY properties [CM88]). In the context of closed systems, we observed the revision problem as the problem of *adding* properties that the original program fails to satisfy, while preserving the existing universally quantified properties of the original program. We studied revision of open systems by considering programs that are subject to a set of uncontrollable *faults* imposed by the environment. We also developed symbolic, distributed, and parallel techniques to improve the performance of our revision algorithms.

Our approach is based on *local revision* of programs whereby behaviors that violate the new properties are removed. This approach also ensures that all existing universally quantified properties continue to be satisfied. As mentioned in the introduction, we classified our results into three types of (1) *sound* and *complete* polynomial-time algorithms, (2) identifying complexity hierarchy, and (3) efficient heuristics that can be deployed in building tools. A sound and complete algorithm is highly valuable since it allows designers to determine whether a given program is *fixable* and if the program is indeed fixable, the algorithm synthesizes a new program that satisfies the desired property (or level of fault-tolerance). The knowledge of complexity bounds is also important in building tools for automated program revision. For instance, an NP-completeness result demonstrates that corresponding tools must utilize efficient heuristics to expedite the revision algorithm at the cost of completeness. Moreover, hardness proofs often identify where the exponential complexity lies in the problem. Thus, thorough analysis of proofs is also crucial in devising efficient heuristics.

In this chapter, we present an overall picture of the status of our research on automated revision of real-time and distributed programs. In Section 16.1, we summarize the contributions of this dissertation. Then, in Section 16.2, we present some open problems and characterize future research directions.

16.1 Contributions

The main results of this dissertation in the context of revising *closed* systems are as follows:

1. We introduced a sound and complete algorithm for adding a single UNITY progress property (i.e., leads-to or ensures) along with a set of UNITY safety properties (i.e., unless, stable, and invariant) to untimed centralized programs. The time complexity of our algorithm is polynomial in the size of the input program's state space.

2. We presented a counterintuitive result where we showed that adding two or more progress properties to an untimed centralized program is NP-complete. Based on this result, we find that adding one progress property is significantly easier than simultaneous addition of multiple such properties.

3. We showed that the problem of adding a single leads-to property to an untimed centralized program while preserving maximum non-determinism is NP-complete.

4. We proved that addition of only one UNITY safety or only one UNITY progress property to a distributed program is NP-complete. This result formally validates the common belief that designing finite state distributed systems is significantly more difficult than finite state centralized systems.

5. We proposed a sound and complete polynomial-time algorithm in the size of the input program's region graph for addition of a single bounded-time leads-to property. Similar to the untimed case, our algorithm is able to add a single bounded-time ensures property along with a set of safety properties as well.

6. We showed another counterintuitive result that the problem of providing maximum non-determinism while adding a single bounded-time leads-to property to a real-time

program is NP-complete (in the size of the input program's region graph) even if the original program satisfies the corresponding unbounded leads-to property.

The main results of this dissertation in the context of revision of *open* systems are as follows:

1. We introduced a generic fault-tolerance framework for real-time programs independent of platform, architecture, and type of faults. In particular, we formally defined what we mean by faults and levels of fault-tolerance (e.g., *soft* and *hard* fault-tolerance) in the context of real-time programs.

2. We presented a polynomial-time algorithm (in the size of the input program's region graph) that adds *bounded-time recovery* from an arbitrary state predicate to another arbitrary state predicate. Using this algorithm, we proposed sound and complete synthesis algorithms for adding nonmasking and soft-masking fault-tolerance to fault-intolerant real-time programs.

3. We introduced a sound and complete algorithm that transforms a fault-intolerant real-time program into a hard-failsafe program where the output program is required to satisfy only one bounded-time response property in the presence of faults.

4. We showed that the problems of adding hard-failsafe (with only one bounded-time response property) and soft-masking fault-tolerance to a fault-intolerant real-time program are PSPACE-complete in the size of the input intolerant program.

5. We showed that the problem of adding hard-masking fault-tolerance is NP-complete in the size of the region graph of the input program.

6. We introduced the notion of bounded-time phased recovery in the context of fault-tolerant real-time programs. We also identified a sufficient condition for existence of a polynomial-time sound and complete algorithm for adding phased recovery to real-time programs.

7. Finally, we showed that the output of our revision algorithms is sensible programs in the sense that any output program can be decomposed to a fault-intolerant program and a set of components called detectors and δ-correctors.

In order to deploy the revision algorithms presented in this dissertation, we first considered a BDD-based implementation of adding a leads-to property to a distributed program (cf. Algorithm 5.1). The performance of our implementation obviously depends on the structure of the input program. In particular, our implementation can handle complex programs with reachable states of size 10^{10} in less than an hour. By a complex program we mean one whose reachability graph involves multiple cycles.

Also, using symbolic techniques, we developed heuristics for addition of fault-tolerance to distributed programs. We demonstrated that synthesis of distributed untimed programs with moderate-sized set of reachable states (10^{50} and beyond) can be achieved within a reasonable amount of time and memory. Our analysis also shows that the growth of the time complexity is sublinear in the state space.

Using our symbolic synthesis algorithms, we developed the tool SYCRAFT for adding fault-tolerance to existing fault-intolerant distributed programs. In SYCRAFT, a distributed fault-intolerant program is specified in terms of a set of processes and an invariant. Each process is specified as its set of transitions, a set of variables that the process can read, and a set of variables that the process can write. Given a set of fault transitions and a safety specification, the tool can synthesize a fault-tolerant distributed program. We demonstrated the application of SYCRAFT in adding fault-tolerance to problems from the literature of distributed computing as well as a data dissemination protocol in sensor networks.

Time and space complexity has always been a major obstacle in deployment of automated formal methods. Thus, exploiting multiple machines to expand available memory and multiple processors to increase computing power seem to be sensible breakthroughs. Thus, as yet another tool to improve the performance of automated revision, we explored the potential of using distributed and parallel techniques in automated program revision. In particular, we proposed a distributed algorithm for synthesizing failsafe and masking untimed centralized programs. We also introduced a symbolic multi-core algorithm for resolving deadlock states in distributed fault-tolerant programs.

16.2 Open Problems and Future Research Directions

In this section, we present some open problems in the context of revision of real-time and distributed programs. We characterize these problems based on their application in (1) complexity theory, (2) techniques for improving the performance of existing algorithms, and (3) deployment of algorithms introduced in this dissertation in synthesis tools in Subsections 16.2.1, 16.2.2, and 16.2.3, respectively. In Subsections 16.2.2 and 16.2.3, we also describe how one can exploit model checking techniques to make synthesis algorithms more efficient. Note, however, that in Subsection 16.2.3, we explain the model checking techniques (e.g., efficient reachability analysis) that can be *trivially* applied in synthesis, but in Subsection 16.2.2, we describe cases where such techniques (e.g., predicate abstraction) cannot be trivially applied due to the natural differences between verification and synthesis.

16.2.1 Open Problems Related to Complexity of Synthesis

Open questions in this category deal with the complexity of decision problems in the context of revision of real-time and distributed programs. Identifying the class of complexity of decision problems is important in the sense that if the problem involves exponential blow-up, a set of new questions on how to cope with such complexity is raised. In this context, the following problems are currently open:

- In Part II of this dissertation, we assumed that all program computations are *unfair*. An important open question is whether involving fair computations changes the complexity hierarchy of algorithms that add progress properties to programs.

- Complexity of synthesis of hard-failsafe fault-tolerant real-time programs where $SPEC_{\overline{br}}$ consists of two bounded-time response properties. (we showed that the problem for only one bounded response property is PSPACE-complete)

- PSPACE-hardness of synthesis of nonmasking and soft-failsafe real-time programs.

- In [KE04], Kulkarni and Ebnenasir address the problem of automated synthesis of *untimed multitolerant* programs, i.e., programs that tolerate multiple classes of faults and provide a (possibly) different level of fault-tolerance to each class. They show that if one needs to add failsafe (respectively, nonmasking) fault-tolerance with respect to one class of faults and masking fault-tolerance with respect to another class of faults, then such addition can be done in polynomial-time in the size of state space of the fault-intolerant program. The novelty of their algorithm is, it adds fault-tolerance in a stepwise fashion. They, however, show that if one needs to add failsafe fault-tolerance with respect to one class of faults and nonmasking fault-tolerance with respect to another class of faults, then the problem is NP-complete. In the context of real-time programs, an open problem is whether stepwise synthesis of multitolerance is feasible. Another future work in this context includes identifying constraints under which a multitolerant real-time program can be designed in a stepwise manner where only one class of faults is considered at a time.

- In Chapter 9, we showed that the problem of adding phased recovery to a program is NP-complete. We also developed a sufficient condition for adding phased recovery in polynomial-time in the size of the input program's region graph. An open problem in this context is developing other sufficient conditions based on the set relation between intermediate and final recovery predicates. A more interesting problem is identifying a necessary condition under which the problem can be solved in polynomial-time.

16.2.2 Open Problems on Improving the Performance of Existing Algorithms

Open questions in this category deal with techniques that improve the performance of existing algorithms. In this context, we note that problems associated with automated formal analysis of distributed and especially real-time systems are generally considered to be difficult. For instance, the simple problem of reachability analysis in timed automata is PSPACE-complete. We plan to investigate the following techniques as a means to cope with the complexity of such problems:

- (*Synthesis using zone automata*) Region graphs are not the most efficient finite representation of real-time programs in terms of space complexity. On the other hand, zone automata [ACH+92] are more efficient time-abstracted representation of real-time programs. For instance, the UPPAAL model checker [LPY97] incorporates zone automata. In this dissertation, since our goal was to evaluate the complexity classes and feasibility of adding fault-tolerance to real-time programs, we focused on region graphs. However, an obvious improvement is modifying the algorithms for revision of real-time programs presented in this dissertation so that they manipulate zone automata rather than detailed region graphs.

- *(Developing efficient heuristics)* A synthesis algorithm with exponential complexity would clearly limit its application only to programs whose state space is small. One way to redress this limitation is to identify a set of heuristics under which the synthesis algorithm takes polynomial time. For instance, let us consider the NP-complete problem of synthesis of hard-masking programs whose safety specifications consist of two bounded response properties. As we showed in the proof of Theorem 8.4.3, the bottleneck of the problem lies in the existence of cycles that permit the program to violate its bounded response properties. In this context, a natural question is under what circumstances we can efficiently identify such undesirable cycles and break them such that the resulting program satisfies the bounded-time response properties. An answer to this question will lead us to heuristics that are more efficient than brute-force solutions.

- *(Multi-processor/Multi-core synthesis algorithms)* In Part IV, we developed two explicit-state approaches for distributed synthesis of failsafe and masking fault-tolerant untimed centralized programs. We also developed a symbolic algorithm for resolving deadlock states in masking fault-tolerant distributed programs. We are currently investigating similar synthesis algorithms in the real-time and symbolic settings. Specifically, as the first step, we are developing a multi-core algorithm (similar to the concept of multi-core saturation [ELS06]) for fault-span generation of programs in synchronous systems.

- *(Predicate abstraction)* Exploiting abstract interpretation techniques is essential for reducing large or infinite state systems to small or finite state systems. While such techniques have been widely used in model checking (e.g., [CC92, SS99]), it is unclear whether existing techniques can be trivially used in the context of program synthesis. Notice that when we represent a state predicate, say X, by a Boolean variable, say x, using an abstraction technique, due to the nature of synthesis, we may need to manipulate X and/or the transitions that originate or end at X. Hence, after this manipulation, x does not represent X any more. Thus, an interesting research direction is to modify existing abstraction techniques or develop new ones that would work in the context of program synthesis and transformation.

- *(Symmetry reduction)* Symmetry reduction is also a means for exploiting the presence of replication in a model, which has yielded considerable success in model checking

[CFJ93, ES96]. Since symmetry occurs very often in distributed programs [AE98], we expect that a significant improvement can be achieved using symmetry reduction techniques.

- *(SAT-based deadlock resolution)* As mentioned in Subsection 11.2, the most serious bottleneck in synthesizing fault-tolerant Byzantine agreement program is deadlock resolution. In particular, we observed that on average, 96% of the total synthesis time is spent in resolving deadlocks. We note that in the case of Byzantine agreement, the deadlock states that are hard to resolve are the ones from where safe recovery to the program invariant is not possible. Such states have to be eliminated. We are currently investigating ways to encode the deadlock resolution problem as a satisfiability problem. Using our encoding, we plan to use a state-of-the-art SMT[1] or SAT solver (e.g., zChaff) as means for benchmark analysis.

  We note that in synthesizing failsafe fault-tolerant versions of distributed programs, where only satisfaction of safety specification in the presence of faults is required, deadlock resolution is not a concern. Thus, we expect that such programs can be synthesized more efficiently.

- *(Symbolic cycle resolution)* Let us consider the following scenario for deadlock resolution. Let σ_d be a deadlock state inside the fault-span $S_\mathcal{P}$ of a program $\mathcal{P}$. Recall that the algorithm Symbolic_Add_Ft (cf. Algorithm 11.1) resolves deadlock states by (1) adding safe recovery transitions, and (2) if such addition is not possible, eliminating the state. Now, suppose that it is not possible to either add safe recovery or eliminate this state (i.e., the resulting program will become empty). In such a case, it may be possible to add a safe transition from σ_d to a state σ_0, where $\sigma_0 \notin S_\mathcal{P}$, and then add another transition from σ_0 to a state σ_1, where $\sigma_1 \in S_\mathcal{P}$ such that recovery from σ_1 to the invariant is possible. In order to ensure that this solution is sound, we need to resolve cycles created by adding transitions that originate from the fault-span and end in a state outside the fault-span. For instance, in our scenario, if there exists the transition (σ_1, σ_d), then we have created the cycle $\sigma_d - \sigma_0 - \sigma_1 - \sigma_d$. Although there exist several heuristics for symbolic cycle *detection* in the literature of model

[1] In SMT (satisfiability modulo theories) solvers (e.g., Yices [yic] and $z3$ [dMB08]), in addition to Boolean variables, one can use other types such as abstract data types, integers, reals etc., in formulae that involve arithmetic and quantifiers as well. We expect that such integration improves the performance of synthesis.

checking [FFK+01, BGS06, GPP03], to the best of our knowledge, there does not exist a solution to symbolic cycle resolution other than our algorithm Add_LeadsTo (cf. Algorithm 5.1). However, this algorithm is not as efficient as our algorithm Symbolic_Add_Ft. One reason is Add_LeadsTo is more general than Symbolic_Add_Ft and it is less likely to fail when there exists a solution. However, an interesting research direction is developing other cycle resolution heuristics that are more efficient than Add_LeadsTo.

Cycle resolution has application in adding properties in untimed centralized programs as well. For instance, the most crucial step in the algorithm Add_UNITY (cf. Section 4.1) is breaking cycles that violate the desirable progress property. Since removing non-progress cycles from programs is often a tedious task, we believe such heuristics will be extremely beneficial if integrated with model checkers.

- *(Dynamic variable reordering)* BDD representation of a Boolean formula is often more space-efficient than an enumerative representation, provided a good ordering of variables is chosen. In model checking, since the goal is to "verify" the correctness of a model against a property, once the BDD of the model is constructed, there is no need to reconstruct it during verification. Hence, an appropriate initial order of variables is sufficient during the course of verification. To the contrary, automated synthesis has a different dynamic, as we often add and remove states and transitions to manipulate a given program such that it satisfies a desired property (e.g., fault-tolerance). In other words, since the structure of a program changes during synthesis, we expect that dynamic reordering of the variables of BDDs will be beneficial. However, there is a trade-off between time spent to reorder variables for memory efficiency on one side, and time spent to synthesize the program on the other side. Thus, another open problem is to determine the circumstances under which dynamic reordering is beneficial.

16.2.3 Extending the Boundaries of SYCRAFT

In this subsection, we describe the model checking techniques that can be trivially applied to our synthesis algorithms in order to improve the efficiency of Sycraft. Moreover, we also describe some applications of our work in model checking.

Improving the Efficiency of SYCRAFT Using Techniques from Model Checking Observe that the complexity of the algorithms presented in this dissertation is comparable (in the same complexity class) to the corresponding problem in model checking. We believe that this complexity order is especially helpful in improving the performance of SYCRAFT by incorporating techniques from model checking.

Based on the analysis of our algorithms and experimental results, we identified five different bottlenecks (depending upon the structure of the program being synthesized), namely, (1) deadlock resolution, (2) computation of fault-span, (3) cycle resolution, (4) computing recovery paths, and (5) checking safety of groups of transitions. The issues of deadlock resolution, cycle resolution, and computing recovery paths are not related to model checking and were discussed in Subsection 16.2.2. On the other hand, computation of fault-span is simply equivalent to computation of the set of states reachable in the presence of faults. Thus, more efficient reachability analysis algorithms are strongly needed to improve the performance of synthesis of fault-tolerant distributed programs. We categorize model checking techniques that can be trivially used in SYCRAFT as follows:

- In our implementation, the Procedure ForwardReachableStates is implemented simply by a next-state relation. This approach is efficient for cases where the size of BDDs are small (e.g., in Byzantine agreement). However, as soon as the size of BDDs become larger (e.g., in token ring), next-state reachability analysis can be as bad as enumerative methods. We are planning to incorporate more recent symbolic techniques such as clustering [RAB$^+$95], partitioning [BCL91], and saturation-based reachability analysis [CLS01, CY05] in our current implementation. These techniques will certainly improve computation of state predicates such as program invariant and fault-span. However, due to the dynamic nature of synthesis, since we add and remove transitions and states, in each iteration of the algorithm, we need to recompute a *new* fault-span starting from the program invariant using the modified set of program transitions. Thus, another improvement is to develop algorithms that reuse the old fault-span from previous iterations and remove unreachable states.

- Observe that in case of Byzantine agreement, the first action of the program never violates safety. This fact suggests that it is beneficial if we can somehow identify such actions and rule them out in early stages of synthesis. Also, observe that if processes of a distributed program are allowed to read and write only a small number of variables (e.g., in token ring), the size of associated group predicates become relatively large. Since violation of safety can be modeled as a satisfiability problem [EK04], we expect that integrating our implementation with SAT and SMT solvers is beneficial.

We believe that the above improvements will enable us to synthesize a large class of fault-tolerant distributed programs from their fault-intolerant version.

Synthesizing Real-Time Programs in SYCRAFT

An important extension of SYCRAFT can be achieved by implementing our algorithm in Chapters 8 and 9. Synthesis of fault-tolerant real-time programs is of special interest, as augmenting programs with the notion of time often makes their formal analysis significantly more difficult.

Benefiting Synthesis in Model Checking

To extend the results of this dissertation, we plan to integrate the algorithms presented in this dissertation with model checking algorithms to provide developers with automated assistance. As a result, if model checking of a program with respect to a property fails, then our revision algorithms automatically (1) determine whether or not the model is *fixable*, and (2) fix the model if it is indeed fixable.

16.3 Other Research Directions

In this section, we describe other interesting research directions in the context of synthesis of fault-tolerant systems. These directions include synthesizing hybrid systems and incorporating multidisciplinary techniques such as machine learning and logic of knowledge.

16.3.1 Synthesizing Fault-Tolerant Hybrid Systems

A few blended theories of analytical and computational models have emerged to capture the hybridity of deeply embedded systems. In particular, the use of *hybrid automata* [Hen96] makes it possible to model both analytical and computational behaviors of embedded systems at the same time. Thus, by developing a framework for defining the concept of levels of fault-tolerance in terms of safe and live semantics of hybrid automata, one can specify and reason about fault-tolerance properties of cyber-physical systems [Lee06, SLMR05] in a formal and elegant fashion. Meanwhile, the fundamental research problem in this context is to identify the possibilities and limitations in developing synthesis algorithms that transform fault-intolerant hybrid systems to fault-tolerant ones. Moreover, the formal framework can be generalized to capture multitolerant hybrid systems as well; building a rich theory of dependable cyber-physical systems.

16.3.2 Incorporating Machine Learning and Data Mining techniques

The input to our transformation problem is a program in terms of a state-transition function and the goal is to generate another system that satisfies a set of new properties. Since state-transition functions can be represented by (possibly weighed) directed graphs, the transformation problem can be formulated as a graph transformation problem as well. In fact, the output is the transformed input graph that satisfies the set of properties of interest.

In recent years, an increasing interest in the use of *graph mining* algorithms has been emerged in the graph transformation community. Interestingly, our transformation problem can be elegantly formulated as a graph mining problem as follows. Having a database of programs (i.e., directed graphs) that satisfy a property of interest and another database containing programs that do not satisfy the property, the problem is to identify possibilities and limitations of deciding the exitance of a solution to the transforation problem for a particular program with respect to a property of interest. Furthermore, if the answer to the decision problem is affirmative, another research problem is to devise algorithms that efficiently find a witness to the decision problem.

16.3.3 Revising Fault-Tolerant Distributed Systems in Epistemic Logic

Another direction is to explore the possibilities and limitations of addition of fault-tolerance to fault-intolerant distributed programs using epistemic logic [FnYMV95]. To this end, first, one develops a knowledge-based formal framework for defining the notions of faults and levels of fault-tolerance in epistemic logic. Unlike the related work, this framework should be generic in the sense that the representation of faults will be possible notwithstanding the *type* of the faults (be they stuck-at, crash, fail-stop, timing, performance, Byzantine, message loss, etc.), the *nature* of the faults (be they permanent, transient, or intermittent), or the ability of the program to observe the *effects* of the faults (be they detectable or undetectable). In this framework, programs will be specified based on the approach introduced by Fagin, Halpern, Moses, and Vardi [FHMV95]. One has to formally define what it means for a knowledge-based program to tolerate a class of faults. The notion of tolerance of a class of faults by a program will be based on an epistemic specification. In other words, a knowledge-based program tolerates a class of faults if and only if it does not violate its epistemic specification in the presence of faults.

It has been shown that synthesis of distributed protocols with more than one agent from epistemic specifications is undecidable [vdMW05]. However, given a knowledge-based program, it is an open question that whether it is possible to revise the program according to some epistemic specification in the absence or presence of faults. If this problem is decidable for all epistemic properties, then it shows that revising knowledge-based programs can be especially useful when it is undecidable to synthesize the given program (i.e., synthesis from scratch is not possible). In case the revision problem is undecidable for arbitrary epistemic specifications, one has to identify epistemic properties based on decidability of their corresponding revision problem.

We would like to note that a knowledge-based formal framework seems to be especially beneficial for specifying and reasoning about programs and protocols with *multiple concerns*, i.e., protocols consist of concerns such as security, fault-tolerance, distribution, communication, real-time, etc. Thus, results on epistemic-based synthesis of fault-tolerance paves the way for further research on specifying and synthesizing protocols with multiple concerns.

Bibliography

[AAE04] P. C. Attie, A. Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(1):125–185, 2004.

[AAMM00] P. M. Alvarez, H. Aydin, D. Mossé, and R. G. Melhem. Scheduling optional computations in fault-tolerant real-time systems. In *Real-Time Computing Systems and Applications (RTCSA)*, 2000.

[ACH+92] R. Alur, C. Courcoubetis, N. Halbwachs, D. L. Dill, and H. Wong-Toi. Minimization of timed transition systems. In *International Conference on Concurrency Theory (CONCUR)*, pages 340–354, 1992.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AE98] P. Attie and E. A. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):51–115, 1998.

[AE01] P.C. Attie and E. A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(2):187 – 242, 2001.

[AFFM06] K. Akesson, M. Fabian, H. Flordal, and R. Malik. Supremica an integrated environment for verification, synthesis and simulation of discrete event systems. In *International Workshop on Discrete Event Systems*, pages 384–385, 2006.

[AFFV03] K. Akesson, M. Fabian, H. Flordal, and A. Vahidi. Supremica a tool for verification and synthesis of discrete event supervisors. In *Mediterranean Conference on Control and Automation*, 2003.

[AFH96] R. Alur, T. Feder, and T.A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, 1996.

[AG93] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[AH93] R. Alur and T.A. Henzinger. Real-Time Logics: Complexity and Expressiveness. *Information and Computation*, 10(1):35–77, May 1993.

[AH97] R. Alur and T. A. Henzinger. Real-time system = discrete system + clock variables. *International Journal on Software Tools for Technology Transfer*, 1(1-2):86–109, 1997.

[AK98a] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, 1998.

[AK98b] A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 436–443, 1998.

[ALW89] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specification. In *International Colloqium on Automata, Languages, and Programming (ICALP)*, pages 1–17, 1989.

[AM99a] P. M. Alvarez and D. Mossé. A responsiveness approach for scheduling fault recovery in real-time systems. In *IEEE Real Time Technology and Applications Symposium (RTAS)*, pages 4–13, 1999.

[AM99b] E. Asarin and O. Maler. As soon as possible: Time optimal control for timed automata. In *Hybrid Systems: Computation and Control (HSCC)*, pages 19–30, 1999.

[AMM00] H. Aydin, R. G. Melhem, and D. Mossé. Optimal scheduling of imprecise computation tasks in the presence of multiple faults. In *Real-Time Computing Systems and Applications (RTCSA)*, pages 289–296, 2000.

[AMP95] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid System*, 1995.

[AMPS98] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *IFAC Symposium on System Structure and Control*, pages 469–474, 1998.

[AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[Att99] P. C. Attie. Synthesis of large concurrent programs via pairwise composition. In *International Conference on Concurrency Theory (CONCUR)*, pages 130–145, 1999.

[ATW05] C. S. Althoff, W. Thomas, and N. Wallmeier. Observations on determinization of büchi automata. In *Implementation and Application of Automata (CIAA)*, pages 262–272, 2005.

[BAK08] B. Bonakdarpour, F. Abujarad, and S. S. Kulkarni. Parallelizing deadlock resolution in symbolic synthesis of distributed programs. Technical Report MSU-CSE-08-25, Department of Computer Science and Engineering, Michigan State University, 2008.

[BBG00] P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the signal environment. In *Discrete Event Dynamic System: Theory and Applications*, pages 325–346, 2000.

[BCD05] P. Bouyer, F. Chevalier, and D. D'Souza. Fault diagnosis using timed automata. In *Foundations of Software Science and Computation Structure*, pages 219–233, 2005.

[BCL91] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In *International Conference on Very Large Scale Integration*, pages 49–58, 1991.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BDMP03] P. Bouyer, D. D'Souza, P. Madhusudan, and A. Petit. Timed control with partial observability. In *Computer Aided Verification (CAV)*, pages 180–192, 2003.

[BEGL99] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. Enhancing model checking in verification by ai techniques. *Artificial Intelligence*, 112:57–104, 1999.

[BEKar] B. Bonakdarpour, A. Ebnenasir, and S. S. Kulkarni. Complexity results in revising UNITY programs. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, To appear.

[BGS06] R. Bloem, H. N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in n log n symbolic steps. *Formal Methods in System Design*, 28(1):37–56, 2006.

[BH00] R. Bharadwaj and C. Heitmeyer. Developing high assurance avionics systems with the SCR requirements method. In *Digital Avionics Systems Conference*, 2000.

[BJG02] J. Bang-Jensen and G. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, 2002.

[BK06a] B. Bonakdarpour and S. S. Kulkarni. Automated incremental synthesis of timed automata. In *International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, LNCS 4346, pages 261–276, 2006.

[BK06b] B. Bonakdarpour and S. S. Kulkarni. Incremental synthesis of fault-tolerant real-time programs. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, LNCS 4280, pages 122–136, 2006.

[BK07a] B. Bonakdarpour and S. S. Kulkarni. Distributed synthesis of fault-tolerant programs in the high atomicity model. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, LNCS 4838, pages 21–36, 2007.

[BK07b] B. Bonakdarpour and S. S. Kulkarni. Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 3–10, 2007.

[BK08a] B. Bonakdarpour and S. S. Kulkarni. Masking faults while providing bounded-time phased recovery. In *International Symposium on Formal Methods (FM)*, pages 374–389, 2008.

[BK08b] B. Bonakdarpour and S. S. Kulkarni. Revising distributed UNITY programs is np-complete. In *Principles of Distributed Systems (OPODIS)*, page To appear, 2008.

[BK08c] B. Bonakdarpour and S. S. Kulkarni. SYCRAFT: A tool for synthesizing fault-tolerant distributed programs. In *Concurrency Theory (CONCUR)*, pages 167–171, 2008.

[BKA08] B. Bonakdarpour, S. S. Kulkarni, and A. Arora. Disassembling real-time fault-tolerant programs. In *ACM International Conference on Embedded Software (EMSOFT)*, page To appear, 2008.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[Car94] A. Carruth. Real-time UNITY. Technical Report CS-TR-94-10, University of Texas at Austin, January 1994.

[CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511547, August 1992.

[CC06] M.-Y. Chung and G. Ciardo. A dynamic firing speculation to speedup distributed symbolic state-space generation. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[CFJ93] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Computer Aided Verification (CAV)*, pages 450–462, 1993.

[CHT96] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[CK89] B. H. C. Cheng and S. M. Kaplan. A semantically oriented program synthesis system. In *Annual Hawaii International Conference on Systems Sciences*, volume 2, pages 85–94, 1989.

[CLS01] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state-space generation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 328–342, 2001.

[CM88] K. M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[CT96] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[CY91] C. Courcoubetis and M. Yannakakis. Minimum and maximum delay problems in real-time systems. In *Computer-Aided Verificaion (CAV)*, pages 399–409, 1991.

[CY05] G. Ciardo and A. J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 146–161, 2005.

[dAFH+03] L. de Alfaro, M. Faella, T. A. Henzinger, R. Majumdar, and M. Stoelinga. The element of surprise in timed games. In *International Conference on Concurrency Theory (CONCUR)*, 2003.

[dAFL06] L. de Alfaro, M. Faella, and A. Legay. An introduction to the tool Ticc. Technical Report UCSC-CRL-06-14, School of Engineering, University of California, Santa Cruz, 2006.

[dAH01] L. de Alfaro and T. A. Henzinger. Interface automata. In *European Software Engineering Conference*, pages 109–120, 2001.

[Dij90] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ., 1990.

[DM02] D. D'Souza and P. Madhusudan. Timed control synthesis for external specifications. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 571–582, 2002.

[dMB08] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[Ebn07] A. Ebnenasir. DiConic addition of failsafe fault-tolerance. In *Automated Software Engineering (ASE)*, pages 44–53, 2007.

[EC82] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[EC06] A. Ebnenasir and B. H. C. Cheng. A pattern-based approach for modeling and analyzing error recovery. In *Workshops on Software Architectures for Dependable Systems (WADS)*, pages 115–141, 2006.

[EC07] A. Ebnenasir and B. H. C Cheng. *Architecting Dependable Systems IV*, chapter A Pattern-Based Approach for Modeling and Analyzing Error Recovery, pages 115–141. Springer Berlin / Heidelberg, 2007.

[EJ91] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *Foundations of Computer Science (FOCS)*, pages 368–377, 1991.

[EK04] A. Ebnenasir and S. S. Kulkarni. SAT-based synthesis of fault-tolerance. Fast Abstracts of the International Conference on Dependable Systems and Networks (DSN), 2004.

[EKAar] Ali Ebnenasir, S. S. Kulkarni, and A. Arora. FTSyn: A framework for automatic synthesis of fault-tolerance. *International Journal of Software Tools for Technology Transfer (STTT)*, To appear.

[EKB05] A. Ebnenasir, S. S. Kulkarni, and B. Bonakdarpour. Revising UNITY programs: Possibilities and limitations. In *On Principles of Distributed Systems (OPODIS)*, pages 275–290, 2005.

[EL86] E. A. Emerson and C. L. Lei. Efficient model checking in fragments of the propositional model mu-calculus. In *Logic in Computer Science (LICS)*, pages 267–278, 1986.

[ELS06] J. Ezekiel, G. Lüttgen, and R. Siminiceanu. Can Saturation be parallelised? on the parallelisation of a symbolic state-space generator. In *International Workshop on Parallel and Distributed Methods of Verification (PDMC)*, pages 331–346, 2006.

[Eme90] E. A Emerson. *Handbook of Theoretical Computer Science*, volume B, chapter 16: Temporal and Modal Logics, pages 995–1067. Elsevier Science Publishers B. V., Amsterdam, 1990.

[Epp99] D. Eppstein. Finding the k shortest paths. *SIAM Journal of Computing*, 28(2):652–673, 1999.

[ES96] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):105–131, August 1996.

[FFK+01] K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *In Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 420–434, 2001.

[FHMV95] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. Knowledge-based programs. In *Symposium on Principles of Distributed Computing (PODC)*, pages 153–163, 1995.

[FHW80] S. Fortune, J. E. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10:111–121, 1980.

[FLM02] M. Faella, S. LaTorre, and A. Murano. Dense real-time games. In *Logic in Computer Science (LICS)*, pages 167–176, 2002.

[FnYMV95] R. Fagin, J.Y. Halpern nad Y. Moses, and M. Vardi. *Reasoning About Knowledge*. The MIT Press, 1995.

[FS07] B. Finkbeiner and S. Schewe. SMT-based synthesis of distributed systems. In *Automated Formal Methods (AFM)*, 2007.

[FW06] L. Feng and W. M. Wonham. TCT: A computation tool for supervisory control synthesis. In *International Workshop on Discrete Event Systems*, pages 388–389, 2006.

[GH00] S. Ghosh and X. He. Fault-containing self-stabilization using priority scheduling. *Information Processing Letters*, 73(3–4):145–151, 2000.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[GJ04] F. C. Gärtner and A. Jhumka. Automating the addition of fail-safe fault-tolerance: Beyond fusion-closed specifications. In *FORMATS/FTRTFT*, pages 183–198, 2004.

[GMS01] H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *8th International SPIN Workshop on Model Checking of Software*, pages 217–234, 2001.

[GPP03] R. Gentilini, C. Piazza, and A. Policriti. Computing strongly connected components in a linear number of symbolic steps. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 573–582, 2003.

[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[GW00] P. Gohari and W. M. Wonham. On the complexity of supervisory control design in the RW framework. *IEEE Transactions on Systems, Man, and Cybernetics*, 30(5):643–652, 2000.

[Hen92] T. A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43(3):135–141, 1992.

[Hen96] T. A. Henzinger. The theory of hybrid automata. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 278–292, 1996.

[HGGS00] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Computer-Aided Verification (CAV)*, pages 20–35, 2000.

[HNSY94] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[JB] B. Jobstmann and R. Bloem. *Lily - A LInear Logic Synthesizer*. http://www.ist.tugraz.at/staff/jobstmann/lily/.

[JB06] B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 117–124, 2006.

[JGB05] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Computer Aided Verification (CAV)*, pages 226–238, 2005.

[JGFS02] A. Jhumka, F. Gartner, C. Fetzer, and N. Suri. On systematic design of fast and perfect detectors. Technical Report 200263, School of Computer and Communication Sciences, EPFL, 2002.

[JGWB07] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *Computer Aided Verification (CAV)*, pages 258–262, 2007.

[JHA07] R. D. Jeffords, C. L. Heitmeyer, and M. Archer. Adding fault-tolerance to requirements specifications. Under review - Personal communication, 2007.

[JHS02] A. Jhumka, M. Hiller, and N. Suri. Component-based synthesis of dependable embedded software. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 111–128, 2002.

[JM06] M. U. Janjua and A. Mycroft. Automatic correction to safety violations in programs. In *Thread Verification (TV)*, page Unpublished, 2006.

[Jur00] Marcin Jurdzinski. Small progress measures for solving parity games. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 290–301, 2000.

[KA00] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 82–93, 2000.

[KA06] S. S. Kulkarni and M. Arumugam. Infuse: A TDMA based data dissemination protocol for sensor networks. *International Journal on Distributed Sensor Networks (IJDSN)*, 2(1):55–78, 2006.

[KAC01] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of Byzantine agreement. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 130–140, 2001.

[KAE07] S. S. Kulkarni, A. Arora, and A. Ebnenasir. *Software Engineering and Fault-Tolerance*, chapter Adding Fault-Tolerance to State Machine-Based Designs. World Scientific Publishing Co. Pte. Ltd, 2007.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In *Symposium on Complexity of Computer Computations*, pages 85–103, 1972.

[KE02] S. S. Kulkarni and A. Ebnenasir. The complexity of adding failsafe fault-tolerance. *International Conference on Distributed Computing Systems (ICDCS)*, pages 337–344, 2002.

[KE03] S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. *International Conference on Distributed Computing Systems*, 2003.

[KE04] S. S. Kulkarni and A. Ebnenasir. Automated synthesis of multitolerance. In *International Conference on Dependable Systems and Networks (DSN)*, pages 209–219, 2004.

[KE05a] S. S. Kulkarni and A. Ebnenasir. Adding fault-tolerance using pre-synthesized components. In *European Dependable Computing Conference (EDCC)*, pages 72–90, 2005.

[KE05b] S. S. Kulkarni and A. Ebnenasir. Complexity issues in automated synthesis of failsafe fault-tolerance. *IEEE Transaction on Dependable and Secure Computing (TDSC)*, 2(3):201–215, 2005.

[KPV06] O. Kupferman, N. Piterman, and M. Y. Vardi. Safraless compositional synthesis. In *International Conference on Computer Aided Verification (CAV)*, pages 31–44, 2006.

[KRS99] S. S. Kulkarni, J. Rushby, and N. Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. In *Internationa Workshop on Self-Stabilization (WSS)*, pages 33–40, June 1999.

[Kul99] S. S. Kulkarni. *Component-based design of fault-tolerance*. PhD thesis, Ohio State University, 1999.

[Lee06] E. A. Lee. Cyber-physical systems - are computing foundations adequate? In *Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*, October 2006.

[LL92] S. Lafortune and F. Lin. On tolerable and desirable behaviors in supervisory control of discrete event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 1(1):61–92, 1992.

[LMM00] F. Liberato, R. G. Melhem, and D. Mossé. Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems. *IEEE Transations on Computers*, 49(9):906–914, 2000.

[LPY97] K. G. Larsen, P.Pattersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[LSW03] M. Leucker, R. Somla, and M. Weber. Parallel model checking for LTL, CTL*, and L_2^μ. In *International Workshop on Parallel and Distributed Model Checking (PDMC)*, 2003.

[LTL90] H.-T. Liaw, J.-H. Tsiah, and C.-S. Lin. Efficient automatic diagnosis of digital circuits. In *Computer-Aided Design (ICCAD)*, pages 464–467, 1990.

[LW90] F. Lin and W. M. Wonham. Decentralized control and coordination of discrete-event systems with partial observation. *IEEE Transactions On Automatic Control*, 35(12), December 1990.

[Mat87] F. Mattern. Algorithms for distributed termination detection. *Journal of Distributed Computing*, 2(3):161–175, 1987.

[MCB89] J. C. Madre, O. Coudert, and J. P. Billon. Automating the diagnosis and the rectification of design error with priam. In *Computer-Aided Design (ICCAD)*, pages 30–33, 1989.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[MMG03] D. Mossé, R. G. Melhem, and S. Ghosh. A nonpreemptive real-time scheduler with recovery from transient faults and its implementation. *IEEE Transactions on Software Engineering*, 29(8):752–767, 2003.

[MNP06] O. Maler, D. Nickovic, and A. Pnueli. From MITL to timed automata. In *Formal Modeling and Analysis of Timed Systems (FORMATS)*, pages 274–289, 2006.

[MW80] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121, 1980.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(1):68–93, 1984.

[MW92] Z. Manna and R. Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, 1992.

[PM98] M. Pandya and M. Malek. Minimum achievable utilization for fault-tolerant processing of periodic tasks. *IEEE Transations on Computers*, 47(10):1102–1112, 1998.

[PPS06] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In

Verification, Model Checking, and Abstract Interpretation (VMCAI), pages 364–380, 2006.

[PR89a] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Principles of Programming Languages (POPL)*, pages 179–190, 1989.

[PR89b] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *International Colloqium on Automata, Languages, and Programming (ICALP)*, pages 652–671, 1989.

[PRS94] D. Paik, S.M. Reddy, and S. Sahni. Deleting vertices to bound path length. *IEEE Transations on Computers*, 43(9):1091–1096, 1994.

[PRS98] D. Paik, S. M. Reddy, and S. Sahni. Vertex splitting in dags and applications to partial scan designs and lossy circuits. *International Journal of Foundations of Computer Science*, 9(4):377–398, 1998.

[PYC94] I. N. Hajj P.-Y. Chung, Y.-M. Wang. Logic design error diagnosis and correction. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2:320332, 1994.

[RAB+95] R. Ranjan, A. Aziz, R. Brayton, B. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. In *IEEE/ACM International Workshop on Logic Synthesis*, 1995.

[RLL03] K. Rudie, S. Lafortune, and F. Lin. Minimal communication in a distributed discrete-event systems. *IEEE Transactions On Automatic Control*, 48(6), June 2003.

[Roh04] K. R. Rohloff. *Computations on distributed discrete-event systems.* PhD thesis, University of Michigan, 2004.

[Rus02] J. Rushby. An overview of formal verification for the time-triggered architecture. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 83–105, 2002.

[RW89] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[RW92] K. Rudie and W. M. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions On Automatic Control*, 37(11):1692–1708, 1992.

[SD97] U. Stern and D. L. Dill. Parallelizing the murφ verifier. In *Computer Aided Verification (CAV)*, pages 256–278, 1997.

[SLMR05] J. A. Stankovic, I. Lee, A. K. Mok, and .R Rajkumar. Opportunities and obligations for physical computing systems. *IEEE Computers*, 38(11):23–31, 2005.

[SLTB+06] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. *ACM SIGPLAN Notices*, 41(11):404–415, 2006.

[Som] F. Somenzi. CUDD: Colorado University Decision Diagram Package. `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`.

[SS83] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computers*, 1(3):222–238, 1983.

[SS99] H. Saïdi and N. Shankar. Abstract and model check while you prove. In *Computer-Aided Verification (CAV)*, pages 443–454, 1999.

[TG99] O Theel and F. Gartner. An exercise in proving convergence through transfer functions. In *Workshop on Self-Stabilizing Systems (SSS)*, pages 41–47, 1999.

[THB95] S. Tasiran, R. Hojati, and R. K. Brayton. Language containment of non-deterministic *omega*-automata. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 261–277, 1995.

[Tho90] W. Thomas. *Handbook of Theoretical Computer Science*, volume B, chapter 4: Automata on Infinite Objects, pages 133–192. Elsevier Science Publishers B. V., Amsterdam, 1990.

[Tho95] W. Thomas. On the synthesis of strategies in infinite games. In *Theoretical Aspects of Computer Science (STACS)*, pages 1–13, 1995.

[Tho02] W. Thomas. Infinite games and verification (extended abstract of a tutorial). In *International Conference on Computer Aided Verification (CAV)*, pages 58–64, 2002.

[Tri02] S. Tripakis. Fault diagnosis for timed automata. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 205–224, 2002.

[vdMW05] R. van der Meyden and T. Wilke. Synthesis of distributed systems from knowledge-based specifications. In *Concurrency Theory (CONCUR)*, pages 562–562, 2005.

[VW86] M.Y. Vardi and P. Wolper. Automata theoretic techniques for modal logic of programs. *Journal of Computer and System Sciences*, 32:183–221, 1986.

[WHT03] N. Wallmeier, P. Hütten, and W. Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In *Implementation and Application of Automata (CIAA)*, pages 11–22, 2003.

[yic] Yices: An SMT Solver. `http://yices.csl.sri.com`.

[YqRzH85] S. Yong-qiang, L. Ru-zhan, and B. Hua. Program synthesis based on boyer-moore theorem proving techniques. In *ACM annual conference on Computer Science*, pages 348–355, 1985.

Appendices

Appendix A

Summary of Notation

$\mathbb{Z}_{\geq 0}$	nonnegative integers
$\mathbb{R}_{\geq 0}$	nonnegative reals
V	set of discrete variables
X	set of clock variables
$\Phi(X)$	set of all clock constraints over X
σ	state
(s, ν)	state specified by location s and clock valuation ν
(σ_0, σ_1)	transition
$(s_0, \nu_0) \rightarrow (s_1, \nu_1)$	transition specified by locations and clock valuations
(s, δ)	delay transition at location s for duration δ
S	state predicate or fault-span
T	transition predicate
T^s	set of immediate transitions in T
T^d	set of delay transitions in T
F	set of fault transitions
V_χ	set of discrete variables of process/program χ
X_χ	set of clock variables of process/program χ
R_χ	set of variables that process/program χ can read
W_χ	set of variables that process/program χ can write
T_χ	transition predicate of process/program χ
$\mathcal{S}_\chi$	state space of process/program χ
$\mathcal{P}$	program

$\mathcal{TC}$	traffic controller program
$\mathcal{BA}$	Byzantine agreement program
$\mathcal{BAFS}$	Byzantine agreement program with fail-stop faults
$\mathcal{TR}$	token ring program
$\mathcal{AS}$	altitude switch program
$\mathcal{IF}$	infuse program
$\Pi_{\mathcal{P}}$	set of processes of program $\mathcal{P}$
$Inv_{\mathcal{P}}$	initial states of program $\mathcal{P}$
$\overline{\sigma}$	computation
$\mathcal{L}$	UNITY property
$\mathcal{B}$	transition predicate that characterizes a set of safety UNITY properties
$SPEC$	specification
$SPEC_e$	existing specification
$SPEC_n$	new specification
$SPEC_{bt}$	set of bad immediate transitions
$SPEC_{\overline{bt}}$	safety specification characterized by $SPEC_{bt}$
$SPEC_{\overline{br}}$	timing dependent safety specification
$\models$	refines
$\models_I$	satisfies from I
ρ	clock region
r	region
(s, ρ)	region specified by location s and clock region ρ
$\mapsto$	leads-to
$\mapsto_{\leq}$	bounded leads-to/bounded response
D	detection predicate
W	witness predicate
C	correction predicate
$\mathcal{D}$	detector
$\mathcal{C}$	δ-corrector
$\setminus$	variable abstrator
$\bullet$	fusion operator
$\oplus$	exclusive-or operator

$[]$	transition insertion or non-determinist execution operator
$\|\|$	simultaneous statements operator
$\triangleright$	S-computation
$\Box\Diamond Q$	always eventually Q