The Dissertation Committee for Roopsha Samanta
certifies that this is the approved version of the following dissertation:

Program Reliability through Algorithmic Design and Analysis

Committee:

E. Allen Emerson, Supervisor

Vijay K. Garg, Supervisor

Aristotle Arapostathis

Adnan Aziz

Joydeep Ghosh

Warren A. Hunt, Jr.

Program Reliability through Algorithmic Design and Analysis

by

Roopsha Samanta, B.E., M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2013

For Ma and Baba,

who inspired me to start a Ph.D.,

and for Naren,

who helped me finish.

Acknowledgments

I thank my advisor, Allen Emerson, for his support on multiple fronts through the years. I will always look back fondly at his insistence on precision and brevity, our spirited technical conversations, and the Turing Award announcement. I thank my co-advisor, Vijay Garg, for his invaluable mentorship, diligent review of my thesis and for being an inspirational role model. I thank the other members of my disseration committee — Ari Arapostathis, Adnan Aziz, Joydeep Ghosh and Warren Hunt — for their help in various stages of this long PhD, and for multiple technical pointers. I am grateful to Gustavo de Veciana, for his guidance, encouragement and refreshingly constructive attitude towards academia.

I treasure each opportunity I have had to collaborate and interact with colleagues outside the university. I am grateful to Swarat Chaudhuri for facilitating our successful collaborations, and for his support and mentorship. I thank Aditya Nori, Sriram Rajamani, Ganesh Ramalingam and Kapil Vaswani at Microsft Research, India for making my internship a truly memorable experience. I am also grateful to Sven Schewe for patiently answering all my questions about his excellent dissertation, and for being an empathetic friend.

I also cherish the friendship, collaboration and guidance I have received from my colleagues at UT Austin over the years. I am grateful to Jyotirmoy

Deshmukh for introducing me to formal methods, for our many gratifying collaborations, and for the hours of animated discussions on technical and not-so-technical topics; Oswaldo Olivo, for being a dedicated collaborator, sounding board and an ever ready dessert buddy; Sandip Ray and Thomas Wahl for their reliable counsel; Bharath Balasubramanium for his continual support; and, Bishwarup Mondal for our enjoyable early collaborations and his support during my transition into formal methods.

I am indebted to Melanie Gulick, who has been an administrative assistant par excellence, and has helped me out countless times, with her typical cheerfulness.

I am deeply grateful to my close friends — Neha, Kaushik, Avani, Ruchir, Bharath and Anitha — for their help and support during the difficult phases, and for lots of happy memories along the way.

My family's support has been relentless, and their patience, incredible. I have lost track of the number of times that my little sister, Poorna, has cooked us dinner, or helped us out with chores, or some member of my extended family has called up to check on a paper notification or my health. My parents and Naren have been with me through every ebb and flow of my time in graduate school, and I cannot imagine this journey without them. I am truly humbled by their love and faith.

Program Reliability through Algorithmic Design and Analysis

Roopsha Samanta, Ph.D.
The University of Texas at Austin, 2013

Supervisors: E. Allen Emerson
Vijay K. Garg

Software systems are ubiquitous in today's world and yet, remain vulnerable to the fallibility of human programmers as well as the unpredictability of their operating environments. The overarching goal of this dissertation is to develop algorithms to enable automated and efficient design and analysis of reliable programs.

In the first and second parts of this dissertation, we focus on the development of programs that are free from programming errors. The intent is not to eliminate the human programmer, but instead to complement his or her expertise, with sound and efficient computational techniques, when possible. To this end, we make contributions in two specific domains.

Program debugging — the process of fault localization and error elimination from a program found to be incorrect — typically relies on expert human intuition and experience, and is often a lengthy, expensive part of the program development cycle. In the first part of the dissertation, we target

automated debugging of sequential programs. A broad and informal statement of the (automated) program debugging problem is to *suitably* modify an erroneous program, say $\mathcal{P}$, to obtain a correct program, say $\widehat{\mathcal{P}}$. This problem is undecidable in general; it is hard to formalize; moreover, it is particularly challenging to assimilate and mechanize the customized, expert programmer intuition involved in the choices made in manual program debugging. Our first contribution in this domain is a methodical formalization of the program debugging problem, that enables automation, while incorporating expert programmer intuition and intent. Our second contribution is a solution framework that can debug infinite-state, imperative, sequential programs written in higher-level programming languages such as C. Boolean programs, which are smaller, finite-state abstractions of infinite-state or large, finite-state programs, have been found to be tractable for program verification. In this dissertation, we utilize Boolean programs for program debugging. Our solution framework involves two main steps: (a) automated debugging of a Boolean program, corresponding to an erroneous program $\mathcal{P}$, and (b) translation of the corrected Boolean program into a correct program $\widehat{\mathcal{P}}$.

Shared-memory concurrent programs are notoriously difficult to write, verify and debug; this makes them excellent targets for automated *program completion*, in particular, for synthesis of synchronization code. Extant work in this domain has focused on either propositional temporal logic specifications with simplistic models of concurrent programs, or more refined program models with the specifications limited to just safety properties. Moreover,

there has been limited effort in developing adaptable and fully-automatic synthesis frameworks that are capable of generating synchronization at different levels of abstraction and granularity. In the second part of this dissertation, we present a framework for synthesis of synchronization for shared-memory concurrent programs with respect to temporal logic specifications. In particular, given a concurrent program $\mathcal{P}$ composed of synchronization-free processes, $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_K$, and a temporal logic specification ϕ_{spec} describing their expected concurrent behaviour, we generate synchronized processes, $\mathcal{P}_1^s, \ldots, \mathcal{P}_K^s$, such that the resulting concurrent program $\mathcal{P}^s$ satisfies ϕ_{spec}. We provide the ability to synthesize readily-implementable synchronization code based on lower-level primitives such as locks and condition variables. We enable synchronization synthesis of finite-state concurrent programs composed of processes that may have local and shared variables, may be straight-line or branching programs, may be ongoing or terminating, and may have program-initialized or user-initialized variables. We also facilitate expression of safety and liveness properties over both control and data variables by proposing an extension of propositional computation tree logic.

Most program analyses, verification, debugging and synthesis methodologies target traditional correctness properties such as safety and liveness. These techniques typically do not provide a quantitative measure of the sensitivity of a computational system's behaviour to unpredictability in the operating environment. We propose that the core property of interest in reasoning in the presence of such uncertainty is *robustness* — small perturbations to the

operating environment do not change the system's observable behavior substantially. In well-established areas such as control theory, robustness has always been a fundamental concern; however, the techniques and results therein are not directly applicable to computational systems with large amounts of discretized, discontinuous behavior. Hence, robustness analysis of software programs used in heterogeneous settings necessitates development of new theoretical frameworks and algorithms.

In the third part of this dissertation, we target robustness analysis of two important classes of discrete systems — string transducers and networked systems of Mealy machines. For each system, we formally define robustness of the system with respect to a specific source of uncertainty. In particular, we analyze the behaviour of transducers in the presence of input perturbations, and the behaviour of networked systems in the presence of channel perturbations. Our overall approach is automata-theoretic, and necessitates the use of specialized *distance-tracking automata* for tracking various distance metrics between two strings. We present constructions for such automata and use them to develop decision procedures based on reducing the problem of robustness verification of our systems to the problem of checking the emptiness of certain automata. Thus, the system under consideration is robust if and only if the languages of particular automata are empty.

Table of Contents

List of Tables

List of Figures

Part I

Introduction

Chapter 1

Introduction

Software systems today are increasingly complex, ubiquitous and often interact with each other or with the physical world. While their reliability has never been more critical, these systems remain vulnerable to the fallibility of human programmers as well as the unpredictability of their operating environments. The only solution that holds promise is increased usage of ***meta-programs*** *to help analyze, debug and synthesize programs, given a precise characterization of reliable program behaviour.*

1.1 Motivation

Human programmers are fallible. As a natural consequence, programming errors, or *bugs*, are as pervasive as software programs today. Bugs are annoying at best, and more often than not, end up affecting the productivity of both software developers and consumers. Occasionally, bugs lead to system failures with disastrous consequences, especially when they show up in safety-critical, mission-critical or economically-vital applications. Examples include infamous bugs in medical, military, aerospace, financial services and automotive software. One of the first quantitative studies of this problem estimated

the annual cost of software bugs in the US to be 59.5 billion dollars [96]!

Besides human error, software systems often grapple with yet another source of undesirable behaviour. We live in an era where computation does not exist in a vacuum. Most computational systems interact with a user or the physical world, or/and, consist of tightly integrated, interacting software components. An inescapable attribute of such complex, heterogeneous systems is *uncertainty*. For example, real-world images handled by image processing engines are frequently noisy, DNA strings processed by transducers in computational biology may be incomplete or incorrectly sequenced, text processors must account for wrongly spelled keywords, the data generated by sensors in medical devices or automobiles can be corrupted, or the channels between system components in a networked power plant controller may lose data packets. Left unchecked, such uncertainty can wreak havoc and lead to highly unpredictable system behaviour.

Given the ever-increasing sophistication of the operating environments, and the complexity and pervasiveness of software systems, the problem of unreliable program behaviour is likely to persist. The only solution that holds promise is the one provided jointly by the formal methods, programming languages and software engineering communities. Summed up simply, their somewhat paradoxical solution is — less dependence on human developers, and increased usage of *meta-programs* to help *analyze*, *debug* and *synthesize* programs, given a precise characterization of reliable program behaviour. This computational approach to development of reliable programs has proven to be

effective in many domains [59, 71, 104, 121]. However, much work remains to be done.

A typical process of ensuring correctness in existing or evolving software code, involves an iterative cycle of error detection, fault localization and error elimination, that ultimately converges to a correct program. Tools based on model checking [27], static program analyses [51] or dynamic program analyses [118] have helped automate the process of error detection to a large extent. However, the process of fault localization and error elimination — also termed as debugging — continues to rely on expert human intuition and experience, and is often a lengthy, expensive part of the program development cycle. While the error detection tools typically provide *error traces* to substantiate their bug discoveries, these traces are usually too long to be inspected manually in any efficient manner. They may be encumbered with unnecessary data and may provide little insight into the actual location of the error within the system.

There is an evident need for simplifying and automating the essentially manual process of debugging as much as possible. The study in [96] indicated that 22.2 billion dollars of the estimated (59.5 billion dollars) annual cost of software bugs could be eliminated by earlier and more effective identification and removal of software defects. In today's multi-core era, software systems as well as hardware designs are increasingly more complex, errors more subtle, and productivity more critical. Debugging is a bigger challenge than ever before. And lest we forget, human debuggers are just as prone to err as human programmers; there always remains the possibility that a bug fix leads to the

introduction of new bugs!

An alternate strategy for developing correct programs is program synthesis. The promise of program synthesis is alluring — skip the iterative error detection and debugging loop, and instead generate correct-by-construction programs from specifications of program behaviour and correctness. Unsurprisingly, in contrast to automated program debugging, synthesis of programs from specifications of user intent has always been a popular research area [47, 59, 102, 120]. The correctness specifications considered have been diverse, ranging from temporal logic or functional specifications to partial programs or input-output examples.

Unfortunately, while program synthesis is an exciting theoretical problem, it has failed to live up to its promise in general. There are niche areas such as logic synthesis in processor design, synthesis of bit-vector programs, synthesis of standard undergraduate textbook algorithms, e.g., sorting, graph algorithms, mutual exclusion, dining philosophers etc., in which program synthesis has indeed been successful. However, synthesizing a large piece of software code from scratch is a computationally intensive task and mandates writing a very rich and detailed specification. There are far too many design choices: programming language, data structures, synchronization primitives, program size, memory usage etc. This necessitates the presence of an expert user to initialize the *synthesizer* with the desired parameters and specifications. As a consequence of all this, in practice, it is often hard to justify synthesizing software code from scratch over modifying legacy code. The best compromise

between the theoretical promise and the practical reality of program synthesis might lie in harnessing synthesis tools for program *completion*; the more modest goal here is to focus on partially written programs, and automatically synthesize missing components, that are particularly tricky for a human programmer to get right.

Most program analyses, verification, debugging and synthesis methodologies target traditional correctness properties such as safety and liveness. These are *qualitative* assertions about individual program executions, and typically do not provide a *quantitative* measure of the sensitivity of a computational system's behaviour to uncertainty in the operating environment. For instance, a program may have a *correct* execution on every individual input, but its output may be highly sensitive to even the minutest perturbation in its inputs. We propose that the core property of interest in reasoning in the presence of uncertainty is *robustness* — small perturbations to the operating environment do not change the system's observable behavior substantially. This property is *differential* in the sense that it relates a range of system executions possible under uncertainty. Development of robust systems demands a departure from techniques to develop traditionally correct systems, as the former requires quantitative reasoning about the system behavior. Given the above, formal reasoning about robustness of systems is a problem of practical as well as conceptual importance.

The ultimate goal of this dissertation is to enable efficient design and analysis of reliable programs. The intent is not to eliminate the human pro-

grammer, but instead to complement his or her expertise, with sound and efficient computational techniques, when possible. To accomplish this goal, we target three specific domains:

1. Debugging of sequential programs,
2. Synchronization synthesis for concurrent programs, and
3. Robustness analysis of discrete systems.

The choice of these domains is not arbitrary. The first two domains focus on bridging the gap between manual program writing combined with debugging, and fully automatic synthesis-from-scratch, for traditional correctness properties. As described earlier, the process of ensuring correctness in manually written software code, involves an iterative cycle of error detection, followed by debugging. This paves the way for a natural separation of concerns for automated tools. It is, thus, reasonable to assume that the input to a debugging engine is a program, that has been found to be incorrect by an error detection engine. We argue that the availability of an incorrect sequential program as input is easier to justify than the availability of an incorrect concurrent program. Despite significant advances in software verification and analyses techniques that have made it possible to detect bugs in many software programs, it is still easier to detect bugs in sequential programs than in concurrent programs. Concurrent programs, with their interacting components, are notorious for exhibiting *heisenbugs* [70] that are difficult to detect, reproduce and of course debug. Thus, automated debugging of sequential programs

is, arguably, a more plausible goal than automated debugging of concurrent programs.

Concurrent programs, on the other hand, are excellent targets for automated synthesis, in particular, for synthesis of synchronization code. Insidious synchronization errors are often the source of common concurrency bugs such as deadlocks, races, starvation and resource access violations. It is conceivable that one can simplify the design and analysis of (shared-memory) concurrent programs by, first, manually writing *synchronization-free* concurrent programs, followed by, automatically synthesizing the synchronization code necessary for ensuring the programs' correct concurrent behaviour.

Our third domain targets robustness analysis of discrete software systems, modeled using transducers, with respect to various sources of uncertainty. Since development of robust software systems is a nascent area, in this dissertation, we focus on formalizing the notion of robustness and developing algorithms to verify robustness of certain classes of discrete systems. We believe this exercise will help us analyze a larger class of systems, and develop synthesis or perhaps debugging tools for guaranteeing robust behaviour in the future.

In what follows, we present an overview of the particular challenges faced in each of these domains, our overall approach for tackling these challenges, and an outline of this dissertation's chapters.

1.2 Debugging of Sequential Programs

Challenges. Historically, research in formal and semi-formal methods for debugging programs (sequential or concurrent) has not garnered as much mainstream interest as research in error detection, verification or even synthesis. Perhaps the problem has been both underrated, as well as perceived to be unsuitable for automation. Indeed, the debugging problem is hard to formalize. Limiting our attention to sequential programs, there can be multiple types of programming errors — arithmetic errors (division by zero, arithmetic overflow or underflow, lack of arithmetic precision due to rounding), non-termination (infinite loops, infinite recursion), conceptual errors, syntax errors, type errors etc. There can be multiple ways to fix a bug — changing the program locally, disallowing certain program inputs, clever usage of key data structures etc. Local changes to a program can also be done in several ways — insertion, deletion, modification or swapping of statements. Thus, one cannot expect a uniform formulation of or solution to the debugging problem. Moreover, it is particularly challenging to assimilate and mechanize the customized, expert human intuition involved in the choices made in manual program debugging.

A broad and informal statement of the (automated) program debugging problem is to compute a correct program $\widehat{\mathcal{P}}$ that is obtained by *suitably* modifying an erroneous program $\mathcal{P}$. Since this requires computation of a correct program, and the problem of verification of an arbitrary program is undecidable, the problem of program debugging is also undecidable in general.

Our Approach. This dissertation presents a methodical formulation and solution framework for the program debugging problem that strives to address the above challenges. We assume that, along with an erroneous program $\mathcal{P}$, we are given a set $\mathcal{U}$ of *update schemas*. An update schema compactly describes a class of permissible modifications of a statement in $\mathcal{P}$; for example, the update schema `assign` $\mapsto$ `assign` permits replacement of an assignment statement with any another assignment statement, and can be applied to the assignment statement $x := y$ to get other assignment statements such as $x := x + y$, $y := x + 1$ etc. Update schemas enable us to refine the formulation of the program debugging problem — the goal is now to compute a correct program $\widehat{\mathcal{P}}$ that is obtained from $\mathcal{P}$ by applying suitable update schemas from $\mathcal{U}$ to the statements in $\mathcal{P}$. Observe that, while a typical debugging routine begins with fault localization and is followed by error elimination, our update schema-based formulation obviates the need for a separate fault localization phase. We directly focus on error elimination by identifying program statements that are *repairable*, i.e., may be modified by applying a permissible update schema in order to obtain a correct $\widehat{\mathcal{P}}$.

Our solution framework targets imperative, sequential programs written in higher-level programming languages such as C. As stated earlier, the verification and debugging problems for such programs in undecidable in general. Verification tools such as SLAM [13], SLAM2 [10], BLAST [65] and SATABS [26] routinely use *predicate abstraction* [56] to generate abstract programs, called *Boolean programs*, for analyzing such infinite-state programs.

Boolean programs, which are equivalent in expressive power to pushdown systems, enjoy nice computational properties such as decidability of reachability and termination [12]. Thus, Boolean programs are more tractable for verification, and as we demonstrate in this dissertation, tractable for debugging as well.

Thus, besides $\mathcal{P}$ and $\mathcal{U}$, our framework also requires a Boolean program $\mathcal{B}$ which demonstrates the incorrectness of $\mathcal{P}$ with an abstract error trace feasible in $\mathcal{P}$. Given all these inputs, our predicate abstraction-based solution framework for computation of a suitable repaired program $\widehat{\mathcal{P}}$ involves two main steps: (a) automated repair of the Boolean program $\mathcal{B}$ using permissible update schemas to get $\widehat{\mathcal{B}}$, and (b) *concretization* of $\widehat{\mathcal{B}}$ to obtain $\widehat{\mathcal{P}}$.

Dissertation Layout. We present our treatment of debugging of sequential programs in Part II of this dissertation.

In Chapter 2, we begin by presenting the syntax, semantics and relevant correctness specifications of a simplified, C-like programming language. We then review the technique of predicate abstraction-refinement and describe Boolean programs. We end with a formal description of the program debugging problem and an overview of our solution framework.

In Chapter 3, we present a simple and efficient algorithm for repairing a class of Boolean programs that meet some syntactic requirements. In this chapter, program correctness is specified using a *precondition* and a *postcondi-*

tion, describing the initial states and final states, respectively, of the program. The algorithm draws on standard techniques from predicate calculus to obtain annotations for the program statements, by *propagating* the given precondition and postcondition through each program statement. These annotations are then used to inspect program statements for repairability and compute a repair if possible. We show that our algorithm can guarantee the correctness and termination of the resulting Boolean program, and can always find a suitable repaired Boolean program, under certain assumptions.

The method presented in Chapter 3 is efficient, but cannot handle programs with arbitrary recursion. Moreover, the method focuses on Boolean programs which can be repaired by modifying exactly one statement using an update schema from $\mathcal{U}$.

In Chapter 4, we generalize the method presented in Chapter 3 in several ways. We present a framework that can handle programs with arbitrary recursion, and can generate repaired programs by modifying multiple statements of the original erroneous program. In this chapter, program correctness is specified using *assertions* included in the program body, specifying the desired states at various points of program execution. Our framework can handle multiple assertions — even if only a single assertion violation was detected in the original program, the computed repaired program is guaranteed to satisfy *all* assertions. Our framework is *cost-aware* — given a user-defined cost function, that charges each application of an update schema to a program statement some user-defined cost, and a repair budget, the framework com-

putes a repaired program whose total modification cost does not exceed the repair budget; we postulate that this cost-aware formulation, along with update schemas, is a flexible and convenient way of incorporating expert programmer intent and intuition in automated program debugging. Besides presenting a generalized algorithm for repairing Boolean programs in Chapter 4, we also describe strategies to translate the repairs computed in Boolean programs to corresponding repairs for the original C program. In particular, we include strategies to ensure the readability of the repaired program using user-defined expression templates. These two steps together enable predicate abstraction-based repair of infinite-state programs. We demonstrate the efficacy of our overall framework by repairing C programs using a prototype tool.

We conclude Part II with a discussion of related work in Chapter 5.

1.3 Synchronization Synthesis for Concurrent Programs

Challenges. Early approaches to synthesis of synchronization for concurrent programs were first developed in [47, 91]. These papers focused on propositional temporal logic specifications and restricted models of concurrent programs such as *synchronization skeletons*. Synchronization skeletons that suppress data variables and data manipulations are often inadequate abstractions of real-world concurrent programs. While finite-state real-world programs can, in principle, be tackled using propositional temporal logic, it can be quite cumbersome to express properties over functions and predicates of program variables using propositional temporal logic. Recent synthesis approaches use

more sophisticated program models and permit specifications involving data variables. However, these approaches are typically applicable only for safety properties, and entail some possibly restrictive assumptions. For instance, it is almost always assumed that all data variables are initialized within the program to specific values, thereby disallowing any kind of user or environment input to a concurrent program. The presence of local data variables is also rarely accounted for or treated explicitly. Finally, there has been limited effort in developing adaptable synthesis frameworks that are capable of generating synchronization at different levels of abstraction and granularity.

Our Approach. In this dissertation, we present a flexible framework for synthesis of synchronization for shared-memory concurrent programs with respect to temporal logic specifications, which generalizes the approach of [47] in several ways. We provide the ability to synthesize more readily-implementable synchronization code based on lower-level primitives such as locks and condition variables. We also enable synchronization synthesis for more general programs and properties.

Given a concurrent program $\mathcal{P}$ composed of synchronization-free processes, $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_K$, and a temporal logic specification ϕ_{spec} describing the expected concurrent behaviour, our goal is to obtain synchronized processes, $\mathcal{P}_1^s, \ldots, \mathcal{P}_K^s$, such that the concurrent program $\mathcal{P}^s$ resulting from their *asynchronous composition* satisfies ϕ_{spec}. This is effected in several steps in our proposed approach. The first step involves automatic generation of a tempo-

ral logic formula $\phi_{\mathcal{P}}$ given $\mathcal{P}_1, \ldots, \mathcal{P}_K$, specifying the concurrency and operational semantics of these unsynchronized processes. The second step involves construction of a tableau T_ϕ, for ϕ given by $\phi_{\mathcal{P}} \wedge \phi_{spec}$. If the overall specification is found to be satisfiable, the tableau yields a global model $\mathcal{M}$, based on $\mathcal{P}_1, \ldots, \mathcal{P}_K$ such that $\mathcal{M} \models \phi$. The next step entails decompositon of $\mathcal{M}$ into the desired synchronized processes $\mathcal{P}_1^s, \ldots, \mathcal{P}_K^s$ with synchronization in the form of *guarded commands* or *conditional critical regions* $CCRs$. The final step involves a correctness-preserving mechanical compilation of the synthesized guarded commands into synchronization code based on lower-level primitives such as locks and condition variables.

Dissertation Layout. We present our approach to synthesis of synchronization for concurrent programs in Part III of this dissertation.

In Chapter 6, we present our first generalization of the approach of [47]. Similar to [47], the framework synthesizes synchronization skeletons given a propositional temporal logic specification of their desired concurrent behaviour. However, in addition to synthesizing high-level synchronization actions in the form of guarded commands, the proposed framework has the ability to perform a correctness-preserving mechanical compilation of guarded commands into synchronization code based on lower-level primitives such as locks and condition variables. We provide the ability to synthesize coarse-grained synchronization code with a single lock for controlling access to shared variables and condition variables, or fine-grained synchronization code with

separate locks for controlling access to shared variables and condition variables. It is up to the user to choose an appropriate granularity of atomicity that suitably balances the trade-off between concurrency and overhead for a particular application/system architecture. This is an important feature, as programmers often restrict themselves to using coarse-grained synchronization for its inherent simplicity. In fact, manual implementations of synchronization code using wait/notify operations on condition variables are particularly hard to get right in the presence of multiple locks. We establish the correctness of both translations — guarded commands to coarse-grained synchronization and guarded commands to fine-grained synchronization — with respect to a useful subset of propositional computation temporal logic (CTL) that includes both safety and liveness properties.

We further establish soundness and completeness of the compilation, and use our prototype tool to successfully synthesize synchronization code for concurrent Java programs such as an airport ground traffic simulator program, readers-writers and dining philosophers.

In Chapter 7, we generalize the approach of [47] to a richer class of programs and specifications. We present a framework that supports finite-state concurrent programs composed of processes that may have local and shared variables, may have a linear or branching control-flow and may be ongoing or terminating. We focus on programs that are *closed systems*, without any interaction with an external environment once execution begins. Note that these include programs in which an environment or user sets the initial values

of the program variables. We propose an extension to propositional CTL that facilitates expression of safety and liveness properties over control and data variables, such as $\mathsf{AG}\,(v_1 = \mu \Rightarrow \mathsf{AF}(loc_2 = \ell_2 \wedge v_2 = \mu + 1))$ (if at any point in an execution, the value of variable v_1 is μ, it is inevitable that control in process $\mathcal{P}_2$ reaches location ℓ_2, wherein the value of variable v_2 is $\mu + 1$).

We describe our adaptation of the tableau construction algorithm for propositional CTL to enable handling our extended specification language. When there exist environment-initialized variables, we present an initial brute-force solution for modifying the basic approach to ensure that $\mathcal{P}^s$ satisfies ϕ_{spec} for all possible initial values of such variables. Also, we address the effect of local variables on the permitted behaviours in $\mathcal{P}^s$ due to limited observability of global states, and discuss solutions. Finally, we adapt the compilation presented in Chapter 6 to synthesize synchronization based on locks and condition variables for this richer class of programs and specifications.

We conclude Part III with a discussion of related work in Chapter 8.

1.4 Robustness Analysis of Discrete Systems

Challenges. In well-established areas such as control theory, robustness has always been a fundamental concern; in fact, there is an entire sub-area of control theory — *robust control* [78] — that extensively studies this problem. However, as robust control typically involves reasoning about continuous state-spaces, the techniques and results therein are not directly applicable to computational systems with large amounts of discretized, discontinuous be-

havior. Moreover, uncertainty in robust control refers to differences between mathematical models and reality; thus robust control focuses on designing controllers that function properly in the presence of perturbation in various system parameters (as opposed to perturbation in the inputs or in the internal channels of a networked system).

Robustness analysis of discrete systems, such as finite-state transition systems, has only recently begun to gain attention. Various notions of robustness for transducers were first proposed in a series of papers [22, 24, 88]. These papers mainly reason about programs that manipulate numbers (as opposed to strings or sequences of numbers). While there is emerging work in quantitative formal reasoning about the robustness of cyber-physical systems, the problem of reasoning about robustness with respect to errors in *networked communication* has been largely ignored. This is unfortunate as communication between different computation nodes is a fundamental feature of many modern systems. In particular, it is a key feature in emerging cyber-physical systems [98, 99] where runtime error-correction features for dealing with uncertainty may not be an option.

Our Approach. In this dissertation, we target robustness analysis of two important classes of discrete systems — string transducers and networked systems of Mealy machines. For each system, we formally define robustness of the system with respect to a specific source of uncertainty. In particular, we analyze the behaviour of transducers in the presence of input perturbations, and

the behaviour of networked systems in the presence of channel perturbations.

Our overall approach is automata-theoretic, and necessitates the use of specialized *distance-tracking automata* for tracking various distance metrics between two strings. We present constructions for such automata and use them to develop decision procedures based on reducing the problem of robustness verification of our systems to the problem of checking the emptiness of certain automata. Thus, the system under consideration is robust if and only if the language of a particular automaton is empty.

Dissertation Layout. We present our approach to robustness analysis of discrete systems in Part IV of this dissertation.

In Chapter 9, we define the transducer models and distance metrics considered in Part IV. Since some of our automata are *reversal-bounded counter machines*, we review such machines in this chapter. Finally, we present constructions for various distance-tracking automata required in Part IV.

In Chapter 10, we target robustness analysis of string transducers with respect to input perturbations. A function encoded as a transducer is defined to be robust if for each small (i.e., bounded) change to any input string, the change in the transducer's output is proportional to the change in the input. Changes to input and output strings are quantified using weighted generalizations of the Levenshtein and Manhattan distances over strings. Our (automata-theoretic) decision procedures for robustness verification with re-

spect to the generalized Manhattan and Levenshtein distance metrics are in PSPACE and EXPSPACE, respectively. For transducers that are Mealy machines, the decision procedures given these metrics are in NLOGSPACE and PSPACE, respectively.

In Chapter 11, we target robustness analysis of networked systems, when the underlying network is prone to errors. We model such a system $\mathcal{N}$ as a set of processes that communicate with each other over a set of internal channels, and interact with the outside world through a fixed set of input and output channels. We focus on network errors that arise from channel perturbations, and assume that we are given a worst-case bound $\boldsymbol{\delta}$ on the number of errors that can occur in the internal channels of $\mathcal{N}$. We say that the system $\mathcal{N}$ is $(\boldsymbol{\delta}, \boldsymbol{\varepsilon})$-robust if the deviation of the output of the perturbed system from the output of the unperturbed system is bounded by $\boldsymbol{\varepsilon}$.

We study a specific instance of this problem when each process is a Mealy machine, and the distance metric used to quantify the deviation from the desired output is either the Manhattan or the Levenshtein distance. We present efficient automata-theoretic decision procedures for $(\boldsymbol{\delta}, \boldsymbol{\varepsilon})$-robustness for both distance metrics.

We conclude Part IV with a discussion of related work in Chapter 12.

Part II

Sequential Program Debugging

In Chapter 1, we have seen that there is a need for automating the essentially manual process of program debugging as much as possible. We have also noted that sequential programs are more credible candidates for automated program debugging than concurrent programs, and have identified some of the challenges in the area.

In this part of the dissertation, we present a formal definition and solution framework for the problem of automated debugging of sequential programs. Our problem formulation obviates the need for a separate fault localization phase, and instead, directly focuses on error elimination or program repair. Our solution framework targets imperative, sequential programs written in programming languages such as C, and is based on predicate abstraction. We assume that we are given an erroneous concrete program $\mathcal{P}$, a corresponding Boolean program $\mathcal{B}$ which exhibits an abstract error trace feasible in $\mathcal{P}$ and a set of permissible update schemas $\mathcal{U}$. Given these inputs, we first compute a correct Boolean program $\widehat{\mathcal{B}}$ by applying a set of permissible update schemas to $\mathcal{B}$. We then concretize the repaired Boolean program to obtain a correct concrete program $\widehat{\mathcal{P}}$.

In Chapter 2, we lay the groundwork by fixing the syntax, semantics and correctness specifications of a simplified, C-like programming language. We review the technique of predicate abstraction-refinement and define Boolean programs. We present a formal definition of the program debugging problem and an overview of our solution framework. In Chapter 3, we focus on the first step of our overall approach: we present a simple and efficient

algorithm targeting Boolean programs that meet some syntactic requirements and can be repaired by a single application of an update schema from $\mathcal{U}$. We show that our algorithm can guarantee the correctness and termination of the repaired Boolean program. In Chapter 4, we generalize our Boolean program repair approach to repair arbitrary Boolean programs using simultaneous applications of multiple update schemas. We also present strategies for concretization of repaired Boolean programs to enable predicate abstraction-based repair of concrete programs. Finally, we conclude Part II with a discussion of related work in Chapter 5.

Chapter 2

Groundwork

Overview. In this chapter, we begin by presenting the syntax, semantics and correctness of a simplified, C-like programming language. We review the technique of predicate abstraction-refinement which can be used for model checking programs written in this language. We then describe Boolean programs, which are abstract programs obtained through predicate abstraction-refinement. We conclude the chapter with a formal definition of the program repair problem and an overview of our solution framework.

2.1 Sequential Programs

2.1.1 Program Syntax

For the technical presentation in this part of the dissertation, we fix a simplified syntax for sequential programs. A partial definition of the syntax is shown in Fig. 2.1. In the syntax, v denotes a variable, $\langle type \rangle$ denotes the type of a variable, F denotes a procedure, ℓ denotes a statement label or location, $\langle expr \rangle$ denotes a well-typed expression, and $\langle bexpr \rangle$ denotes a Boolean-valued expression.

Thus, a (sequential) program consists of a declaration of global vari-

$\langle pgm \rangle$	::=	$\langle vardecl \rangle \, \langle proclist \rangle$
$\langle vardecl \rangle$	::=	`decl` $v : \langle type \rangle$`;` $\mid \langle vardecl \rangle \, \langle vardecl \rangle$
$\langle proclist \rangle$	::=	$\langle proc \rangle \, \langle proclist \rangle \mid \langle proc \rangle$
$\langle proc \rangle$	::=	$F(v_1, \ldots, v_k)$ `begin` $\langle vardecl \rangle \, \langle stmtseq \rangle$ `end`
$\langle stmtseq \rangle$	::=	$\langle labstmt \rangle$ `;` $\langle stmtseq \rangle$
$\langle labstmt \rangle$	::=	$\langle stmt \rangle \mid \ell : \langle stmt \rangle$
$\langle stmt \rangle$	::=	`skip` $\mid v_1, \ldots, v_m$ `:=` $\langle expr_1 \rangle, \ldots, \langle expr_m \rangle$
		$\mid$ `if` $(\langle bexpr \rangle)$ `then` $\langle stmtseq \rangle$ `else` $\langle stmtseq \rangle$ `fi`
		$\mid$ `while` $(\langle expr \rangle)$ `do` $\langle stmt \rangle$ `od` $\mid$ `assume` $(\langle bexpr \rangle)$
		$\mid$ `call` $F(\langle expr_1 \rangle, \ldots, \langle expr_k \rangle) \mid$ `return`
		$\mid$ `goto` ℓ_1 `or` ... `or` $\ell_n \mid$ `assert` $(\langle bexpr \rangle)$

Figure 2.1: Programming language syntax

ables, followed by a list of procedure definitions; a procedure definition consists of a declaration of local variables, followed by a sequence of (labeled) statements; a statement is a `skip`, (parallel) assignment, conditional, loop, `assume`, (call-by-value) procedure `call`, `return`, `goto` or `assert` statement.

We make the following assumptions: (a) there is a distinguished initial procedure `main`, which is not called from any other procedure, (b) all variable and formal parameter names are globally unique, (c) the number of actual parameters in a procedure call matches the number of formal parameters in the procedure definition, (d) `goto` statements are not used arbitrarily; they are used only to simulate the flow of control in structured programs, (e) the last statement in the loop body of every `while` statement is a `skip` statement, and (e) $\langle type \rangle$ includes integers and Booleans.

Note that the above syntax does not permit return values from procedures. However, return values can be easily modeled using extra global

variables. Hence, this syntax simplification does not affect the expressivity of the programming language. Indeed, the above syntax is quite general.

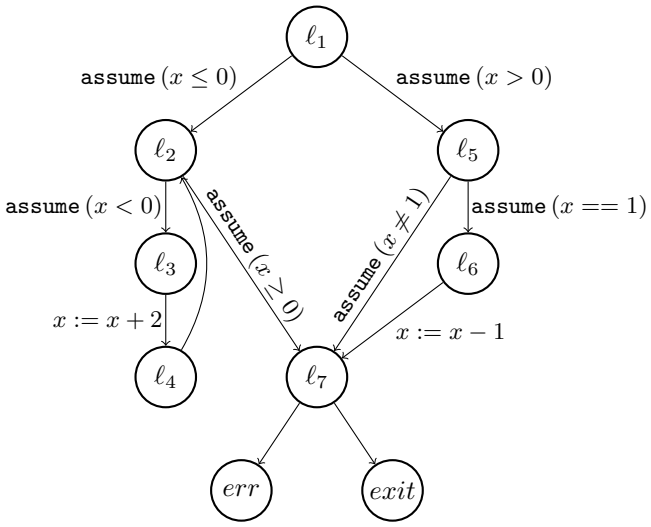
Notation. Let us fix some notation before we proceed. For program $\mathcal{P}$, let $\{F_0, \ldots, F_t\}$ be its set of procedures with F_0 being the `main` procedure, and let $GV(\mathcal{P})$ denote the set of global variables. For procedure F_i, let S_i and $\mathcal{L}_i$ denote the sets of statements and locations, respectively, and let FV_i and LV_i denote the sets of formal parameters and local variables, respectively, with $FV_i \subseteq LV_i$. Let $V(\mathcal{P}) = GV(\mathcal{P}) \cup \bigcup_{i=1}^{t} LV_i$ denote the set of variables of $\mathcal{P}$, and $\mathcal{L}(\mathcal{P}) = \bigcup_{i=1}^{t} \mathcal{L}_i$ denote the set of locations of $\mathcal{P}$. For a location ℓ within a procedure F_i, let $inscope(\ell) = GV(\mathcal{P}) \cup LV_i$ denote the set of all variables in $\mathcal{P}$ whose scope includes l. We denote by $stmt(\ell)$, $formal(\ell)$ and $local(\ell)$ the statement at ℓ and the sets of formal parameters and local variables of the procedure containing ℓ, respectively. We denote by $entry_i \in \mathcal{L}_i$ the location of the first statement in F_i. When the context is clear, we simply use V, $\mathcal{L}$ instead of $V(\mathcal{P})$, $\mathcal{L}(\mathcal{P})$ etc.

2.1.2 Transition Graphs

In addition to a *textual* representation, we will often find it convenient to use a *transition graph* representation of programs, as shown in the example in Fig. 2.2. The transition graph representation of $\mathcal{P}$, denoted $\mathcal{G}(\mathcal{P})$, comprises a set of labeled, rooted, directed graphs $\mathcal{G}_0, \ldots, \mathcal{G}_t$, which have exactly one node, err, in common. Informally, the i^{th} graph $\mathcal{G}_i$ captures the flow of control in procedure F_i with its nodes and edges labeled by locations and corresponding

```
main() {
   int x;
   ℓ1 : if (x ≤ 0)
   ℓ2 :    while (x < 0){
   ℓ3 :      x := x + 2;
   ℓ4 :      skip;
             }
        else
   ℓ5 :    if (x == 1)
   ℓ6 :      x := x − 1;
   ℓ7 : assert (x > 1);
}
```

(a) $\mathcal{P}$



(b) $\mathcal{G}(\mathcal{P})$

Figure 2.2: An example program and its transition graph

statements of F_i, respectively. To be more precise, $\mathcal{G}_i = (N_i, Lab_i, E_i)$, where the set of nodes N_i, given by $\mathcal{L}_i \cup exit_i \cup err$, includes a unique entry node $entry_i$, a unique exit node $exit_i$ and the error node err, the set of labeled edges $E_i \subseteq N_i \times Lab_i \times N_i$ is defined as follows: for all $\ell, \ell' \in N_i$, $(\ell, \varsigma, \ell') \in E_i$ iff:

- $stmt(\ell)$ is an assignment, `assume` (g) or `call` $F(e_1, \ldots, e_k)$ statement, ℓ' is the next sequential location[1] in F_i after ℓ and $\varsigma = stmt(\ell)$, or,

- $stmt(\ell)$ is a `skip` statement and either (a) $stmt(\ell)$ is the last statement in the loop body of a statement ℓ' : `while` (g) and ς is the empty label,

[1]The next sequential location of the last statement in the `then` or `else` branch of a conditional statement is the location following the conditional statement. The next sequential location of the last statement in the `main` procedure is $exit_0$.

or, (b) ℓ' is the next sequential location in F_i after ℓ and ς is the empty label, or,

- $stmt(\ell)$ is $\texttt{if}\,(g)$, and either (a) ℓ', denoted $Tsucc(\ell)$, is the location of the first statement in the `then` branch and $\varsigma = \texttt{assume}\,(g)$, or, (b) ℓ', denoted $Fsucc(\ell)$, is the location of the first statement in the `else` branch and $\varsigma = \texttt{assume}\,(\neg g)$, or,

- $stmt(\ell)$ is $\texttt{while}\,(g)$, and either (a) ℓ', denoted $Tsucc(\ell)$, is the location of the first statement in the `while` loop body and $\varsigma = \texttt{assume}\,(g)$, or, (b) ℓ', denoted $Fsucc(\ell)$, is the next sequential location in F_i after the end of the `while` loop body and $\varsigma = \texttt{assume}\,(\neg g)$, or,

- $stmt(\ell)$ is $\texttt{assert}\,(g)$, and either ℓ', denoted $Tsucc(\ell)$, is the next sequential location in F_i after ℓ and ς is the empty label, or, (b) ℓ', denoted $Fsucc(\ell)$, is the node err and ς is the empty label, or,

- $stmt(\ell)$ is a `goto` statement that includes the label ℓ', and ς is the empty label, or,

- $stmt(\ell)$ is a `return` statement, $\ell' = exit_i$ and $\varsigma = \texttt{return}$.

Let $succ(\ell)$ denote the set $\{\ell' : (\ell, \varsigma, \ell') \in E_i\}$ for some $i \in [0, t]$.

A *path* π in $\mathcal{G}_i$ is a sequence of labeled connected edges; with some overloading of notation, we denote the sequence of statements labeling the edges in π as $stmt(\pi)$. Not that every node in $\mathcal{G}_i$ is on some path between $entry_i$ and $exit_i$.

2.1.3 Program Semantics

Given a set $V_s \subseteq V$ of variables, a *valuation* Ω of V_s is a function that maps each variable in V_s to an appropriate value of its *type*. Ω can be naturally extended to map well-typed expressions over variables to values.

An operational semantics can be defined for our programs by formalizing the effect of each type of program statement on a program *configuration*. A configuration η of a program $\mathcal{P}$ is a tuple of the form (ℓ, Ω, ζ), where $\ell \in \bigcup_{i=0}^{t} N_i$, Ω is a valuation of the variables in $inscope(\ell)$[2] and ζ is a stack of elements. Each element of ζ is of the form $(\widetilde{\ell}, \widetilde{\Omega})$, where $\widetilde{\ell} \in \mathcal{L}_i$ for some i and $\widetilde{\Omega}$ is a valuation of the variables in $local(\widetilde{\ell})$. A program *state* is a pair of the form (ℓ, Ω), where ℓ and Ω are as defined above; thus a program state excludes the stack contents. A configuration (ℓ, Ω, ζ) of $\mathcal{P}$ is called an initial configuration if $\ell = entry_0$ is the entry node of the `main` procedure and ζ is the empty stack. We use $\eta \rightsquigarrow \eta'$ to denote that $\mathcal{P}$ can transition from configuration $\eta = (\ell, \Omega, \zeta)$ to configuration $\eta' = (\ell', \Omega', \zeta')$; the transitions rules for each type of program statement at ℓ and for exit nodes of procedures are presented in Fig. 2.3.

Let us take a closer look at the last two transition rules in Fig. 2.3 - the only transition rules that affect the stack contents. Upon *execution* of the statement `call` $F_j(e_1, \ldots, e_k)$ in program configuration (ℓ, Ω, ζ), control moves to the entry node of the called procedure F_j; the new valuation Ω' of

[2]For $\ell = exit_i$, $inscope(\ell) = GV \cup LV_i$, and for $\ell = err$, $inscope(\ell)$ is undefined.

Cases		$(\ell, \Omega, \zeta) \rightsquigarrow (\ell', \Omega', \zeta')$ if:
$stmt(\ell)$:	`skip` `return`	$\ell' = succ(\ell)$, $\Omega' = \Omega$ and $\zeta' = \zeta$
	`goto` ℓ_1 `or` ... `or` ℓ_n	$\ell' \in succ(\ell)$, $\Omega' = \Omega$ and $\zeta' = \zeta$
	`assume` g	$\Omega(g) =$ `true`, $\ell' = succ(\ell)$, $\Omega' = \Omega$ and $\zeta' = \zeta$
	`if` g `while` g	either $\Omega(g) =$ `true`, $\ell' = Tsucc(\ell)$, $\Omega' = \Omega$ and $\zeta' = \zeta$, or, $\Omega(g) =$ `false`, $\ell' = Fsucc(\ell)$, $\Omega' = \Omega$ and $\zeta' = \zeta$
	`assert` g	either $\Omega(g) =$ `true`, $\ell' = Tsucc(\ell)$, $\Omega' = \Omega$ and $\zeta' = \zeta$, or, $\Omega(g) =$ `false` and $\ell' = Fsucc(\ell) = err$
	$v_1, \ldots, v_m :=$ $e_1, \ldots, e_m$	$\ell' = succ(\ell)$, $\forall i \in [1, m] : \Omega'(v_i) = \Omega(e_i)$, $\forall v \notin \{v_1, \ldots, v_m\} : \Omega'(v) = \Omega(v)$ and $\zeta' = \zeta$
	`call` $F_j(e_1, \ldots, e_k)$	$\ell' = entry_j$, $\forall v_i \in formal(\ell') : \Omega'(v_i) = \Omega(e_i)$, $\forall v \in GV(\mathcal{P}) : \Omega'(v) = \Omega(v)$ and $\zeta' = (succ(\ell), \Delta).\zeta$, where $\forall v \in local(\ell) : \Delta(v) = \Omega(v)$
ℓ:	$exit_j$	$\ell' = \ell_{ret}$, $\forall v \in local(\ell') : \Omega'(v) = \Delta(v)$, $\forall v \in GV(\mathcal{P}) : \Omega'(v) = \Omega(v)$ and $\zeta = (\ell_{ret}, \Delta).\zeta'$

Figure 2.3: Transition rules for $(\ell, \Omega, \zeta) \rightsquigarrow (\ell', \Omega', \zeta')$.

program variables is constrained to agree with Ω on the values of all global variables, and maps the formal parameters of F_j to the values of the actual arguments according to Ω; finally, the element $(succ(\ell), \Delta)$ is pushed onto the stack, where $succ(\ell)$ is the location to which control returns after F_j completes execution and Δ is a valuation of all local variables of the calling procedure, as recorded in Ω. The last transition rule in Fig. 2.3 captures the return of control to the calling procedure, say F_c, after completion of execution of a called procedure, say F_j; the top of the stack element (ℓ_{ret}, Δ) is removed and is used to retrieve the location ℓ_{ret} of F_c to which control must return as well the valuation Δ of the local variables of F_c; the new valuation Ω' of program

variables is constrained to agree with Ω on the values of all global variables, and to agree with Δ on the values of all local variables of F_c.

An *execution path* of program $\mathcal{P}$ is a sequence of configurations, $\eta \rightsquigarrow \eta' \rightsquigarrow \eta'' \rightsquigarrow \ldots$, obtained by repeated application of the transition rules from Fig. 2.3, starting from an initial configuration η. Note that an execution path may be finite or infinite. The last configuration (ℓ, Ω, ζ) of a finite execution path may be a *terminating configuration* with $\ell = exit_0$, or an *error configuration* with $\ell = err$ or a *stuck configuration*. An execution path ends in a stuck configuration η if none of the transition rules from Fig. 2.3 are applicable to η. In particular, notice that a transition from configuration (ℓ, Ω, ζ) with $stmt(\ell)$ being `assume` (g) is defined only when $\Omega(g) =$ `true`.

2.1.4 Specifications and Program Correctness

In this part of the dissertation, a specification for a sequential program $\mathcal{P}$ is either (a) a *precondition*, *postcondition* pair, or (b) a set of *assertions*. A precondition, denoted φ, represents the expected set of initial program states, and is a quantifier-free, first order expression capturing the initial values of the program variables in $inscope(entry_0)$. A postcondition, denoted ψ, represents the set of desired final states upon completion of execution of $\mathcal{P}$, and is a quantifier-free, first order expression relating the initial and final values of the program variables in $inscope(entry_0)$. An assertion in $\mathcal{P}$, is a statement of the form ℓ : `assert` (g), with g being a quantifier-free, first order expression representing the expected values of the program variables in $inscope(\ell)$ at ℓ.

We will use the term assertion to denote both the statement $\ell : \texttt{assert}\,(g)$ as well as the expression g. We say a program configuration (ℓ, Ω, ζ) satisfies a precondition, postcondition or assertion, if the embedded variable valuation Ω satisfies the same.

Given a program $\mathcal{P}$ and a precondition, postcondition pair (φ, ψ), $\mathcal{P}$ is said to be *partially correct* iff the Hoare triple $\{\varphi\}\mathcal{P}\{\psi\}$ is *valid*, i.e., iff every *finite* execution path of $\mathcal{P}$, begun in an initial configuration satisfying φ, ends in a terminating configuration satisfying ψ. $\mathcal{P}$ is said to be *totally correct* iff the Hoare triple $\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$ is valid, i.e., iff every execution path, begun in an initial configuration satisfying φ, is finite and ends in a terminating configuration satisfying ψ.

Given a program $\mathcal{P}$ annotated with a set of assertions, $\mathcal{P}$ is partially correct iff every *finite* execution path of $\mathcal{P}$ ends in a terminating configuration. $\mathcal{P}$ is totally correct iff every execution path is finite and ends in a terminating configuration.

Unless otherwise specified, an *incorrect* program is one that is not partially correct.

2.2 Predicate Abstraction and Boolean Programs

The problem of program verification - checking correctness of a program or program model with respect to a specification - is an undecidable problem in general. *Model checking* is a fully automatic program verification technique,

initially proposed as a sound and complete algorithm for checking correctness of finite-state program models with respect to propositional temporal logic specifications. Model checking algorithms have since been devised for certain classes of infinite-state programs and various specification languages. The main challenge in model checking algorithms, which can be informally described as sophisticated algorithms for exhaustive graph search, is the *state explosion* problem: this problem essentially refers to the enormous number of program states or configurations that need to be examined for verifying correctness. Thus, even when the underlying problem is decidable, a naive application of model checking may often be intractable.

The term *abstraction* is used to refer to a collection of techniques that enables model checking of infinite-state or large finite-state programs by reducing them to smaller finite-state programs. In this context, the original program is termed the *concrete* program and the reduced program is called the *abstract* program. The basic idea in the reduction is to compute an approximation of the concrete program by omitting certain details, while preserving information relevant for verifying the given specification. A *conservative* abstraction of a concrete program $\mathcal{P}$ is an over-approximation of the behaviour of $\mathcal{P}$ in the sense that for every concrete execution path in $\mathcal{P}$, there exists a corresponding abstract execution path. For certain classes of specifications such as safety properties - *something bad does not happen* - this means that if the abstract program is successfully model checked, the concrete program $\mathcal{P}$ is guaranteed to be correct. However, if the abstract program is found to violate the spec-

ification, the concrete program may or may not be correct. In particular, a *counterexample* path to an error state in the abstract program may be *spurious*,i.e., it may not correspond to an actual path in $\mathcal{P}$. Thus, the goal is to find an abstract program which is precise enough to verify correctness or exhibit a non-spurious counterexample path, while being amenable to model checking. The standard solution is *counterexample-guided abstraction refinement* (CEGAR), which iteratively *refines* an initial abstraction by analyzing and eliminating spurious abstract counterexample paths.

2.2.1 Predicate Abstraction

Predicate abstraction is an effective abstraction technique for model checking infinite-state sequential programs with respect to safety properties. This technique computes a finite-state, conservative abstraction of a concrete program $\mathcal{P}$ by partitioning $\mathcal{P}$'s state space based on the valuation of a finite set $\{\phi_1, \ldots, \phi_r\}$ of predicates. The resulting abstract program is termed a *Boolean program* $\mathcal{B}$: the control-flow of $\mathcal{B}$ is the same as that of $\mathcal{P}$ and the set $\{b_1, \ldots, b_r\}$ of variables of $\mathcal{B}$ are Boolean variables, with each b_i representing the predicate ϕ_i for $i \in [i, r]$. Given a concrete program $\mathcal{P}$, the overall method proceeds as follows. In step one, an initial Boolean program $\mathcal{B}$ is computed and in step two, $\mathcal{B}$ is model-checked with respect to its specification. If $\mathcal{B}$ is found to be correct, the method concludes that $\mathcal{P}$ is correct. Otherwise, an abstract counterexample path leading to some violated assertion in $\mathcal{B}$ is computed and examined for feasibility in $\mathcal{P}$. If found feasible, the method ter-

minates, reporting an error in $\mathcal{P}$. If found infeasible, in step three, $\mathcal{B}$ is refined into a new Boolean program $\mathcal{B}'$ that eliminates the spurious counterexample. Thereafter, steps two and three are repeated, as needed. Note that the overall method is incomplete - it may not always be able to possible to compute a suitable refinement that eliminates a spurious counterexample or to check if an abstract counterexample is indeed spurious.

In our work, the interesting case is when the method terminates reporting an error. Henceforth, we fix a concrete program $\mathcal{P}$, and a corresponding Boolean program $\mathcal{B}$ that exhibits a non-spurious counterexample path. Let $\{\phi_1, \ldots, \phi_r\}$ denote the set of predicates used in the abstraction of $\mathcal{P}$ into $\mathcal{B}$, where each predicate is a quantifier-free order expression over the variables of $\mathcal{P}$. Let $\{b_1, \ldots, b_r\}$ denote the corresponding Boolean variables of $\mathcal{B}$. Let γ denote the mapping of Boolean variables to their respective predicates: for each $i \in [1, r]$, $\gamma(b_i) = \phi_i$. The mapping γ can be extended in a standard way to expressions over the Boolean variables.

2.2.2 Boolean Programs

Boolean programs (see Fig. 2.4b for a Boolean program corresponding to the program in Fig. 2.2a, generated using SATABS [26]) are sequential programs with a syntax similar to that in Fig. 2.1, with two main differences: (a) all variables and formal parameters are Boolean, and (b) all expressions - $\langle expr \rangle$, $\langle bexpr \rangle$ - are Boolean expressions defined as follows:

$$
\begin{array}{lcl}
\langle bexpr \rangle & ::= & * \mid \langle detbexpr \rangle \\
\langle detbexpr \rangle & ::= & \texttt{true} \mid \texttt{false} \mid b \\
& & \mid \neg \langle detbexpr \rangle \mid \langle detbexpr \rangle \Rightarrow \langle detbexpr \rangle \\
& & \mid \langle detbexpr \rangle \vee \langle detbexpr \rangle \mid \langle detbexpr \rangle \wedge \langle detbexpr \rangle \\
& & \mid \langle detbexpr \rangle = \langle detbexpr \rangle \mid \langle detbexpr \rangle \neq \langle detbexpr \rangle,
\end{array}
$$

where b is a Boolean variable. Thus, a Boolean expression is either a deterministic Boolean expression or the expression $*$, which nondeterministically evaluates to `true` or `false`[3]. We assume that $*$ expresses a *fair* nondeterministic choice, i.e., $*$ does not permanently evaluate to the same value. We assume that Boolean expressions in `assume` $(\langle bexpr \rangle)$ and `assert` $(\langle bexpr \rangle)$ statements are always deterministic.

The transition graph and operational semantics of Boolean programs can be defined in a manner similar to that of concrete sequential programs (see Fig. 2.4 for a Boolean program and its transition graph representation). The main modifications are as follows. In defining the set of labeled edges E_i of graph $\mathcal{G}_i = (N_i, Lab_i, E_i)$ in the transition graph representation $\mathcal{G}(\mathcal{B})$ of $\mathcal{B}$, for $\ell \in N_i$ with $stmt(\ell)$ given by `if` $(*)$ or `while` $(*)$, $Tsucc(\ell)$, $Fsucc(\ell)$ are defined as in Sec. 2.1.2, but the labels ς_1, ς_2 in $(\ell, \varsigma_1, Tsucc(\ell))$, $(\ell, \varsigma_2, Fsucc(\ell))$ are each set to `assume` $(\texttt{true})$. For $stmt(\ell)$ given by `if` $(*)$ or `while` $(*)$, we say $(\ell, \Omega, \zeta) \rightsquigarrow (\ell', \Omega', \zeta')$ if $\ell' \in succ(\ell)$, $\Omega' = \Omega$ and $\zeta' = \zeta$. For $stmt(\ell)$

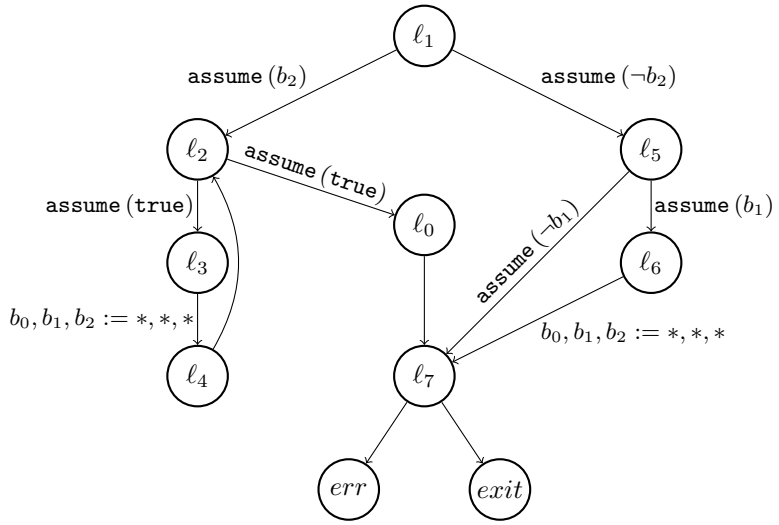
[3]In practice, a nondeterministic Boolean expression is any Boolean expression containing $*$ or the expression `choose`(e_1, e_2), with e_1, e_2 being deterministic Boolean expressions (if e_1 is `true`, `choose`(e_1, e_2) evaluates to `true`, else if e_2 is `true`, `choose`(e_1, e_2) evaluates to `false`, else `choose`(e_1, e_2) evaluates to $*$). While we handle arbitrary nondeterministic Boolean expressions in our prototype tool (see Sec. 4.4), we only consider $*$ expressions in our exposition for simplicity.

```
main() {
   int x;
   ℓ1 : if (x ≤ 0)
   ℓ2 :    while (x < 0){
   ℓ3 :      x := x + 2;
   ℓ4 :      skip;
           }
        else
   ℓ5 :    if (x == 1)
   ℓ6 :      x := x − 1;
   ℓ7 : assert (x > 1);
}
```

(a) $\mathcal{P}$

```
main() {
   /*γ(b0) = x ≤ 1, γ(b1) = x == 1, γ(b2) = x ≤ 0*/
   Bool b0, b1, b2 := *, *, *;
   ℓ1 : if (¬b2) then goto ℓ5;
   ℓ2 : if (*) then goto ℓ0;
   ℓ3 : b0, b1, b2 := *, *, *;
   ℓ4 : goto ℓ1;
   ℓ0 : goto ℓ7;
   ℓ5 : if (¬b1) then goto ℓ7;
   ℓ6 : b0, b1, b2 := *, *, *;
   ℓ7 : assert (¬b0);
}
```

(b) $\mathcal{B}$



(c) $\mathcal{G}(\mathcal{B})$

Figure 2.4: An example concrete program $\mathcal{P}$, a corresponding Boolean program $\mathcal{B}$ and $\mathcal{B}$'s transition graph

given by the assignment statement $b_1, \ldots, b_j, \ldots, b_m := e_1, \ldots, *, \ldots, e_m$, we say $(\ell, \Omega, \zeta) \rightsquigarrow (\ell', \Omega', \zeta')$ if $\ell' = succ(\ell)$, $\zeta' = \zeta$, $\forall i \in \{1, \ldots, j-1, j+1, \ldots, m] : \Omega'(b_i) = \Omega(e_i)$, $\forall v \notin \{b_1, \ldots, b_m\} : \Omega'(v) = \Omega(v)$, and either $\Omega'(b_j) = \texttt{true}$ or $\Omega'(b_j) = \texttt{false}$. This transition rule can be extended to handle other scenarios such as assignment statements with multiple $*$ expressions in the RHS, and `call` statements with $*$ expressions in the actual arguments.

In specifying correctness for Boolean programs, we interpret the nondeterminism in them as Dijkstra's *demonic* nondeterminism [36]. Thus, given a program $\mathcal{B}$ and a precondition, postcondition pair (φ, ψ), $\mathcal{B}$ is said to be partially correct iff every *finite* execution path, begun in an initial configuration satisfying φ ends in a terminating configuration satisfying ψ, for *all* nondeterministic choices that $\mathcal{B}$ might make. $\mathcal{B}$ is said to be totally correct iff every execution path, begun in an initial configuration satisfying φ, is finite and ends in a terminating configuration satisfying ψ, for all nondeterministic choices that $\mathcal{B}$ might make. Given a program $\mathcal{B}$ annotated with a set of assertions, $\mathcal{B}$ is partially correct iff every *finite* execution path of $\mathcal{B}$ ends in a terminating configuration for all nondeterministic choices that $\mathcal{B}$ might make. $\mathcal{B}$ is totally correct iff every execution path is finite and ends in a terminating configuration, for all nondeterministic choices that $\mathcal{B}$ might make.

2.3 Program Repair: The Problem

Let Σ denote the set of *statement types* in program $\mathcal{P}$. For example, for programs with just assignment and `skip` statements, $\Sigma = \{\texttt{assign}, \texttt{skip}\}$.

Given a statement s, let $\tau(s)$ be an element of Σ denoting the statement type of s. Let $\mathcal{U} = \{u_0, u_1, \ldots, u_d\}$ be a set of permissible, statement-level *update schemas*: $u_0 = id$ is the *identity* update schema that maps every statement to itself, and u_i, $i \in [1, d]$, is a function $\sigma \mapsto \widehat{\sigma}$, $\sigma, \widehat{\sigma} \in \Sigma \setminus \{\texttt{assert}\}$, that maps a statement type to a statement type. For each update schema u, given by $\sigma \mapsto \widehat{\sigma}$, we say u can be *applied* to statement s to get statement $\widehat{s}$ if $\tau(s) = \sigma$; $\tau(\widehat{s})$ is then given by $\widehat{\sigma}$. For example, u, given by $\texttt{assign} \mapsto \texttt{assign}$, can be applied to the assignment statement $\ell : x := y$ to get other assignment statements such as $\ell : x := x + y$, $\ell : y := x + 1$ etc. Notice that update schemas in $\mathcal{U}$ do not affect the label of a statement, and that we do not permit any modification of an `assert` statement.

We extend the notion of a statement-level update to a program-level update as follows. For programs $\mathcal{P}$, $\widehat{\mathcal{P}}$, let the respective sets of locations be $\mathcal{L}$, $\widehat{\mathcal{L}}$ and let $stmt(\ell)$, $\widehat{stmt}(\ell)$ denote the respective statements at location ℓ. Let $\mathbb{R}_{\mathcal{U},\mathcal{L}} : \mathcal{L} \to \mathcal{U}$ be an *update function* that maps each location of $\mathcal{P}$ to an update schema in $\mathcal{U}$. We say $\widehat{\mathcal{P}}$ is a $\mathbb{R}_{\mathcal{U},\mathcal{L}}$-update of $\mathcal{P}$ iff $\mathcal{L} = \widehat{\mathcal{L}}$ and for each $\ell \in \mathcal{L}$, $\widehat{stmt}(\ell)$ is obtained by applying $\mathbb{R}_{\mathcal{U},\mathcal{L}}(\ell)$ on $stmt(\ell)$.

Given an incorrect program $\mathcal{P}$[4] and a set of permissible, statement-level update schemas $\mathcal{U}$, the goal of automated program repair is to compute $\widehat{\mathcal{P}}$ such that:

[4]Since total correctness of programs is generally harder to check and verify than partial correctness, in the problem definition, we accept programs that have been shown to be *not* partially correct. We can obviously also accept programs that are known to be *not* totally correct.

1. $\widehat{\mathcal{P}}$ is correct, and,

2. $\widehat{\mathcal{P}}$ is some $\mathbb{R}_{\mathcal{U},\mathcal{L}}$-update of $\mathcal{P}$.

We emphasize the importance of formulating the program repair problem with respect to a set $\mathcal{U}$ of update schemas. Without such a set, there would be no restriction on the *relation* of the repaired program $\widehat{\mathcal{P}}$ to the incorrect program $\mathcal{P}$; in particular, $\widehat{\mathcal{P}}$ could be any correct program constructed from scratch, without using $\mathcal{P}$ at all. Moreover, the set $\mathcal{U}$, which is provided by the user/developer, helps capture some of the expert programmer intuition about the types of programming errors and fixes to expect and explore in a specific automated program repair application. This not only helps generate a repaired program *similar* to what the programmer may have in mind, but also helps reduce the search space for repaired programs.

2.4 Solution Overview

We present a predicate abstraction-based solution framework for automated repair of a concrete program $\mathcal{P}$. Recall that we had fixed a Boolean program $\mathcal{B}$ such that $\mathcal{B}$ is obtained from $\mathcal{P}$ via iterative predicate abstraction-refinement and $\mathcal{B}$ exhibits a non-spurious counterexample path. Besides $\mathcal{P}$ and $\mathcal{U}$, additional inputs to our framework are: the Boolean program $\mathcal{B}$ and the corresponding function γ that maps Boolean variables to their respective predicates. The computation of a suitable repaired program $\widehat{\mathcal{P}}$ involves two main steps:

1. Automated repair of $\mathcal{B}$ to obtain $\widehat{\mathcal{B}}$, and

2. *Concretization* of $\widehat{\mathcal{B}}$ to obtain $\widehat{\mathcal{P}}$.

The problem of automated repair of a Boolean program $\mathcal{B}$ can be defined in a manner identical to the problem definition in Sec. 2.3. Concretization of $\widehat{\mathcal{B}}$ involves mapping each modified statement of $\widehat{\mathcal{B}}$ into a corresponding statement of $\mathcal{P}$ using the function γ. In the following chapters, we describe different algorithms for tackling each step.

Chapter 3

Repair of Boolean Programs

Overview. In this chapter, we present a simple and efficient algorithm for automatic repair of Boolean programs that meet some syntactic requirements and can be repaired by a single application of an update schema. Our approach targets total correctness with respect to a specification in the form of a precondition, postcondition pair, is sound and is complete under certain assumptions.

3.1 Formal Framework

In this chapter, we fix the following restrictions on the syntax of Boolean programs, described in Sec. 2.2.2: (a) there are no `assume`, `goto` or `assert` statements, and (b) there are no recursive procedure calls. Further, in this chapter, we make a distinction between an `if` statement - `if` (g) - and an entire conditional statement - `if` (g) $S_{\texttt{if}}$ `else` $S_{\texttt{else}}$ `fi`, where $S_{\texttt{if}}$, $S_{\texttt{else}}$ are sequences of (labeled) statements; we denote the latter statement type as `cond`. Similarly, we make a distinction between a `while` statement - `while` (g) - and an entire loop statement - `while` (g) `do` $S_{\texttt{loop}}$ `od`, where $S_{\texttt{loop}}$ is a sequence of (labeled) statements; we denote the latter statement type as `loop`. Thus, the

φ : `true`
$b_1 := b_1 \oplus b_2;$ $b_2 := b_1 \wedge b_2;$ $b_1 := b_1 \oplus b_2;$
$\psi : b_2[end] = b_1[init] \ \wedge \ b_2[end] = b_1[init]$

Figure 3.1: Example Boolean program and specification

set of statement types in the Boolean programs of this chapter is given by: Σ = {`skip`, `assign`, `if`, `cond`, `while`, `loop`, `call`, `return`}.

We assume that program correctness is specified using a precondition, postcondition pair (φ, ψ). In particular, we specify total correctness of a Boolean program $\mathcal{B}$ using the Hoare triple $\langle\varphi\rangle\, \mathcal{B}\, \langle\psi\rangle$. Thus, $\mathcal{B}$ is totally correct iff every execution path, begun in an initial configuration satisfying φ, is finite and ends in a terminating configuration satisfying ψ, for *all* nondeterministic choices that $\mathcal{B}$ might make.

Example. We refer the reader to Fig. 3.1 for a Boolean program which is supposed to swap the values in the variables b_1 and b_2. The program is incorrect.

We fix the following set of permissible, statement-level update schemas for the Boolean programs of this chapter:

$$\mathcal{U} = \{id, \texttt{assign} \mapsto \texttt{skip},\ \texttt{cond} \mapsto \texttt{skip},\ \texttt{loop} \mapsto \texttt{skip},$$
$$\texttt{assign} \mapsto \texttt{assign},\ \texttt{if} \mapsto \texttt{if},\ \texttt{while} \mapsto \texttt{while}\}.$$

A single application of an update schema from $\mathcal{U}$ is thus a deletion of an assignment, conditional or loop statement (i.e., replacement by a `skip` statement), or replacement of an assignment statement with another assignment

statement, or modification of the guard of a conditional or loop statement.

For this chapter, we adapt the problem definition of Sec. 2.3 as follows. Given a precondition, postcondition pair (φ, ψ), a Boolean program $\mathcal{B}$ such that $\{\varphi\}\,\mathcal{B}\,\{\psi\}$ is not valid, and the set $\mathcal{U}$ of update schemas defined above, the goal is to compute $\widehat{\mathcal{B}}$ such that:

1. $\langle\varphi\rangle\,\widehat{\mathcal{B}}\,\langle\psi\rangle$ is valid, and,
2. there exists $\mathbb{R}_{\mathcal{U},\mathcal{L}} : \mathcal{L} \to \mathcal{U}$:
   (a) $\widehat{\mathcal{B}}$ is some $\mathbb{R}_{\mathcal{U},\mathcal{L}}$-update of $\mathcal{B}$, and
   (b) $\mathbb{R}_{\mathcal{U},\mathcal{L}}(\ell) \neq id$ for exactly one $\ell \in \mathcal{L}$.

Thus, in this chapter, we only target Boolean programs that can be repaired by a single application of an update schema. Our solution to the above problem has two main steps.

1. We first annotate the program text by *propagating* φ and ψ through each program statement.
2. We use these annotations to inspect statements for *repairability*, and compute a repair if possible.

In the following sections, we explain each of these steps in detail. For ease of exposition, we first describe our algorithm for a program without any procedure calls, and then outline an extension of the approach to programs with procedure calls in Sec. 3.3.3.

3.2 Step I: Program Annotation

Let $\mathcal{B}$ be a single procedure program with a set $\mathcal{L} = \{1, \ldots, n\}$ of locations and a set $V = \{b_1, \ldots b_r\}$ of Boolean program variables. Propagation of the pre-condition φ and the post-condition ψ through program statements is based on the techniques used for Hoare logic [37, 67]. We denote the precondition φ by $\varphi[0]$, and the preconditions propagated forward through the statements $stmt(1), \ldots, stmt(n)$ by $\varphi[1], \ldots, \varphi[n]$, respectively. Similarly, we denote the postcondition ψ by $\psi[n]$, and the postconditions propagated back through statements $stmt(n), \ldots, stmt(1)$ by $\psi[n-1], \ldots, \psi[0]$, respectively.

To aid efficient computation and storage of propagated preconditions and postconditions, we create copies of the program variables to represent variable valuations at certain program locations of interest. We use the copy $V[init]$ of the program variables to represent the initial variable values and the copy $V[end]$ to represent the variable values after execution of $stmt(n)$. Further, the variables valuations *before* and *after* execution of any statement in the program are represented using the copies $V[curr]$ and $V[next]$, respectively. Henceforth, for clarity, an expression e over $V[curr]$, $V[next]$ is sometimes denoted by $e[curr]$, $e[next]$, respectively.

We refer the reader to Fig. 3.2 for the indexed and annotated version of the Boolean program from Fig. 3.1. Observe that the precondition φ is an expression over $V[init]$, the postcondition ψ is an expression over $V[init]$ and $V[end]$, all propagated preconditions $\varphi[1], \ldots, \varphi[n]$ are expressions over $V[init]$ and $V[next]$ and all propagated postconditions $\psi[n-1], \ldots, \psi[0]$ are

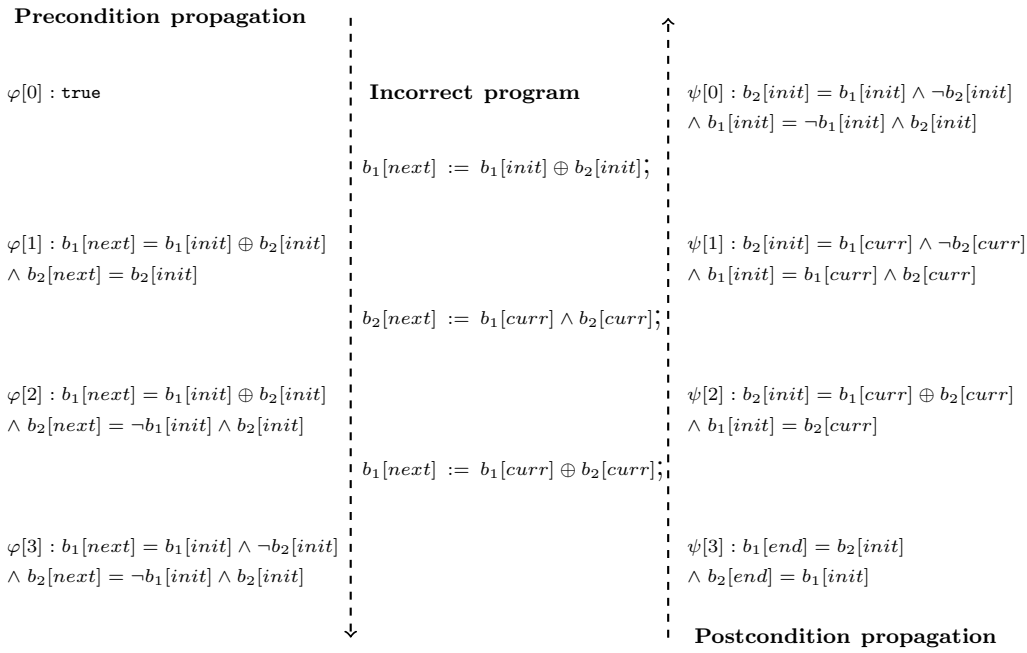


Figure 3.2: Precondition and postcondition propagation

expressions over $V[init]$ and $V[curr]$. We emphasize that our propagation techniques, combined with the absence of $*$ expressions in $\varphi[0]$ and $\psi[n]$, ensure the absence of $*$ expressions in all propagated preconditions and postconditions.

3.2.1 Backward Propagation of Postconditions

Backward propagation of a postcondition $\psi[\ell]$ through $stmt(\ell)$ corresponds to computing $\psi[\ell-1]$ such that the Hoare triple $\langle\psi[\ell-1]\rangle\, stmt\,(\ell)\langle\psi[\ell]\rangle$ is `true`. In other words, $\psi[\ell-1]$ is the set of all variable valuations such that execution of $stmt(\ell)$ begun in any state satisfying $\psi[\ell-1]$ is guaranteed to terminate in a state satisfying $\psi[\ell]$, for all (nondeterministic) choices made

by $stmt(\ell)$. Thus, $\psi[\ell-1]$ is essentially Dijkstra's *weakest precondition* [37], $wp(stmt(\ell), \psi[\ell])$. Given a statement $stmt(\ell)$ and a postcondition $\psi[\ell]$ - an expression over $V[next]$ and $V[init]$ - we compute $\psi[\ell-1] = wp(stmt(\ell), \psi[\ell])$ - an expression over $V[curr]$ and $V[init]$ - using the following inductive rules:

1. `skip`.

   The weakest precondition of $\psi[\ell]$ over a skip statement is essentially the same as $\psi[\ell]$, except each occurrence of variable $b_i[next]$ in $\psi[\ell]$ is replaced with $b_i[curr]$, for all $i \in [1, r]$. Thus, we have:

   $$\psi[\ell-1] = \psi[\ell][\forall i \in [1,r] : b_i[next]/b_i[curr]],$$

   where the notation $\mathcal{A}[b/e]$ represents the expression obtained by replacing all occurrences of b in $\mathcal{A}$ by e.

2. Assignment statement: $b_1, \ldots, b_m := e_1, \ldots, e_m$.

   Let us rewrite this as: $b_1[next], \ldots, b_m[next] := e_1[curr], \ldots, e_m[curr]$. The weakest precondition over an assignment statement is computed by replacing each variable $b_i[next]$ in $\psi[\ell]$ with its assigned expression $e_i[curr]$ for $i \in [1, m]$, and replacing all other variables $b_j[next]$ in $\psi[\ell]$ with $b_j[curr]$. Thus, the resulting expression $\psi[\ell-1]$ is an expression over $V[curr]$ and $V[init]$, given by:

   $$\begin{aligned}\psi[\ell-1] \;=\; \psi[\ell][&\forall i \in [1,m] : b_i[next]/e_i[curr],\\ &\forall i \notin [1,m] : b_i[next]/b_i[curr]].\end{aligned}$$

If for some j, $e_j[curr]$ is the nondeterministic expression $*$, the weakest precondition is computed as the *conjunction* of the weakest preconditions over the statements:

$b_1[next], \ldots, b_m[next]$:= $e_1[curr], \ldots,$ `false`$, \ldots, e_m[curr]$, and

$b_1[next], \ldots, b_m[next]$:= $e_1[curr], \ldots,$ `true`$, \ldots, e_m[curr]$.

For multiple nondeterministic expressions in the RHS of the assignment statement, the weakest precondition can be similarly computed as the conjunction of the weakest preconditions over all possible assignment statements obtained by substituting each $*$ expression with either `false` or `true`.

3. Sequential composition: $stmt(\ell - 1)$; $stmt(\ell))$.

   The weakest precondition over a sequence $stmt(\ell-1)$; $stmt(\ell))$ of statements is computed by first propagating $\psi[\ell]$ through $stmt(\ell)$ to get $wp(stmt(\ell), \psi[\ell])$, followed by propagating $wp(stmt(\ell]), \psi[\ell])$ through $stmt(\ell - 1)$ to obtain the required weakest precondition:

   $$wp(stmt(\ell - 1); stmt(\ell),\ \psi[\ell]) \ =\ wp(stmt(\ell - 1),\ wp(stmt(\ell), \psi[\ell])).$$

   For proper bookkeeping, for each $i \in [1, r]$, all occurrences of $b_i[curr]$ in the expression for $wp(stmt(\ell), \psi[\ell])$ are swapped with $b_i[next]$ before propagating through $stmt(\ell - 1)$.

4. Conditional statement: `if` (g) `then` $S_{\texttt{if}}$ `else` $S_{\texttt{else}}$ `fi`.

   We rewrite this as: `if` $(g[curr])$ `then` $S_{\texttt{if}}$ `else` $S_{\texttt{else}}$ `fi`. The weakest precondition over a conditional statement is given by the the weakest

precondition over the sequence $S_{\texttt{if}}$ of statements if the guard g is `true`, and by the weakest precondition over the sequence $S_{\texttt{else}}$ if g is `false`:

$$\psi[\ell - 1] \;=\; g[curr] \Rightarrow wp(S_{\texttt{if}}, \psi[\ell]) \;\wedge\; \neg g[curr] \Rightarrow wp(S_{\texttt{else}}, \psi[\ell]).$$

If $g = *$, the weakest precondition is given by the *conjunction* of the weakest preconditions over $S_{\texttt{if}}$ and $S_{\texttt{else}}$: $wp(S_{\texttt{if}}, \psi[\ell]) \wedge wp(S_{\texttt{else}}, \psi[\ell])$.

5. Loop statement: `while` (g) `do` $S_{\texttt{loop}}$ `od`.

   The weakest postcondition over a loop statement is computed as a *fix-point* over the weakest preconditions over each loop iteration. We define the weakest precondition of the i^{th} loop iteration, $wp(S^i_{\texttt{loop}}, \psi[\ell])$, as the set of variable valuations such that execution of the loop statement begun in any state satisfying $wp(S^i_{\texttt{loop}}, \psi[\ell])$ is guaranteed to terminate in a state satisfying $\psi[\ell]$ after executing the loop *at most* i times. This is computed as follows:

$$\begin{aligned} wp(S^i_{\texttt{loop}}, \psi[\ell]) &= \bigvee_{j=0}^{i} Y_j, \\ \text{where,} \quad Y_0 &= \psi[\ell] \wedge \neg g, \\ Y_j &= g \wedge wp(S_{\texttt{loop}}, Y_{j-1}). \end{aligned}$$

   Y_0 represents the set of variable valuations that never enter the loop and satisfy $\psi[\ell]$. Y_j represents the set of variable valuations that enter the loop and exit it after *exactly* j loop iterations, terminating in a state satisfying $\psi[\ell]$. To keep the above exposition clean, we have not made the variable copies (curr, next) explicit. Note that each Y_j is computed

in terms of variables in $V[curr]$, which are then swapped with their respective copies in $V[next]$ before computing $wp(S_{\texttt{loop}}, Y_j)$.

Since Boolean programs have a finite number of variable valuations and the above weakest preconditions $wp(S^0_{\texttt{loop}}, \psi[\ell]), wp(S^1_{\texttt{loop}}, \psi[\ell]), \ldots$ are monotonically increasing sets, the Knaster-Tarski Theorem [67] guarantees that this computation terminates in a fixpoint with $wp(S^I_{\texttt{loop}}, \psi[\ell])$ $= wp(S^{I+1}_{\texttt{loop}}, \psi[\ell])$, for some I. The weakest precondition over the loop statement $stmt(\ell)$, which corresponds to all variable valuations which enter the loop and *eventually* exit it in a state satisfying $\psi[\ell]$, is thus given by:

$$\psi[\ell - 1] \;=\; wp(S^I_{\texttt{loop}}, \psi[\ell]) = \bigvee_{j=0}^{I} Y_j.$$

If $g = *$, then we define a different set of iterants Z_j, which correspond to variable valuations that exit the loop in a state satisfying $\psi[\ell]$ in the j^{th} loop iteration when g evaluates to `false`. We remind the reader that $*$ expresses fair choice between `true` and `false`. Hence, g is guaranteed to evaluate to `false` eventually. The weakest precondition, which corresponds to all variable valuations that exit the loop in a state satisfying $\psi[\ell]$ for *all* values of the guard, is given by the fixpoint: $\psi[\ell - 1] = \bigwedge_{j=0}^{I} Z_j$, where, $Z_0 = \psi[\ell]$, $Z_j = wp(S_{\texttt{loop}}, Z_{j-1})$.

In the event that $wp(stmt(\ell), \psi[\ell])$ evaluates to `false` for any statement $stmt(\ell)$, our algorithm aborts propagation of postconditions, and proceeds to the next phase — precondition propagation.

3.2.2 Forward Propagation of Preconditions

Forward propagation of a precondition $\varphi[\ell - 1]$ through a statement $stmt(\ell)$ involves computing $\varphi[\ell]$ such that the Hoare triple $\langle \varphi[\ell - 1] \rangle\, stmt(\ell)\, \langle \varphi[\ell] \rangle$ is `true`. In other words, $\varphi[\ell]$ represents the smallest set of variable valuations such that execution of $stmt(\ell)$, begun in any state satisfying $\varphi[\ell-1]$, is guaranteed to terminate in one of them for all (nondeterministic) choices that $stmt(\ell)$ might make. We call this set the *strongest postcondition* [1], $sp(stmt(\ell), \varphi[\ell-1])$.

In this work, precondition propagation involves recording the value of each program variable in terms of the initial values of the variables, along with the conditions imposed by the program's control-flow. Let us assume that $\varphi[\ell - 1]$ is a Boolean formula of the form[2]:

$$\varphi[\ell - 1] = \rho[init] \wedge \bigwedge_{i \in [1,r]} b_i[curr] = \xi_i[init],$$

where each $\rho[init]$ is a Boolean expression over $V[init]$ representing the *path condition* imposed by the program control-flow and each $\xi_i[init]$ is a Boolean expression over $V[init]$ representing b_i in terms of the initial values of the program variables. In particular, note that if each variable b_i is initialized to some value $initval_i$ in a program, the precondition φ for the program can

[1] Traditionally, the definition of the strongest postcondition (sp) is the dual of the weakest *liberal* precondition (wlp) - these predicate transformers are used in the context of partial correctness and do not concern themselves with program termination [37]. Hence, the traditional sp should ideally be called the strongest *liberal* postcondition. In our work, we define strongest postcondition to be the dual of the weakest precondition for total correctness.

[2] In general, as we will see, $\varphi[\ell - 1]$ is a disjunction over Boolean formulas of this form; $sp(stmt(\ell), \varphi[\ell - 1])$ can then be obtained by computing a disjunction over the strongest postconditions obtained by propagating each such Boolean formula through $stmt(\ell)$.

be written as an expression of this form: $\rho[init] \wedge \bigwedge_{i\in[1,r]} b_i[curr] = initval_i$. Given a statement $stmt(\ell)$ and a precondition $\varphi[\ell-1]$ - an expression over $V[init]$ and $V[curr]$ as above - we compute $\varphi[\ell] = sp(stmt(\ell), \varphi[\ell-1])$ - an expression over $V[init]$ and $V[next]$ of the same form - using the inductive rules enumerated below. In what follows, for an expression $e[curr]$, we find it convenient to use a special notation, $e[curr/init]$ to denote the expression $e[curr][\forall i \in [1,r] : b_i[curr]/\xi_i[init]]$ obtained by substituting each occurrence of variable $b_i[curr]$ in e by its corresponding expression $\xi_i[init]$ from $\varphi[\ell-1]$. Thus, $e[curr/init]$ is an expression over $V[init]$.

1. `skip`.

   The strongest postcondition of $\varphi[\ell-1]$ over a skip statement is given by:

   $$\varphi[\ell] \;=\; \varphi[\ell-1][\forall i \in [1,r] : b_i[curr]/b_i[next]].$$

   Equivalently, $\varphi[\ell] = \rho[init] \wedge \bigwedge_{i\in[1,r]} b_i[next] = \xi_i[init]$.

2. Assignment statement: $b_1, \ldots, b_m := e_1, \ldots, e_m$.

   The strongest postcondition $sp(x{:=}e, \delta)$ for an assignment statement — $x{:=}e$, is given by $\exists y : \delta[x/y] \wedge x = e[x/y]$ [37]. Here, y represents the *unknown* value of x prior to the assignment. This necessitates existential quantification over y to obtain the postcondition. Our use of variable copies enables us to preserve necessary history information and avoid

existential quantification. Thus, we have:

$$\varphi[\ell] \;=\; \rho[init] \wedge \bigwedge_{i \in [1,m]} b_i[next] = e_i[curr] \wedge \bigwedge_{i \notin [1,m]} b_i[next] = b_i[curr].$$

We replace all occurrences of each variable $b_i[curr]$ in the above expression with the corresponding $\xi_i[init]$ expression from $\varphi[\ell - 1]$ to obtain the following equivalent expression for $\varphi[\ell]$ over $V[init]$ and $V[next]$:

$$\varphi[\ell] \;=\; \rho[init] \wedge \bigwedge_{i \in [1,m]} b_i[next] = e_i[curr/init] \wedge \bigwedge_{i \notin [1,m]} b_i[next] = \xi_i[init].$$

For nondeterministic expressions in the RHS of the assignment statement, the strongest postcondition is computed as the *disjunction* of the strongest postconditions over all possible assignment statements obtained by substituting each $*$ expression with either `false` or `true`.

3. Sequential composition: $stmt(\ell - 1)$**;** $stmt(\ell))$.

   The rule for propagation of the precondition $\varphi[\ell - 1]$ through a sequence of statements is similar to the rule for propagation of a postcondition:

   $$sp(stmt(\ell - 1)\textbf{;}\ stmt(\ell),\ \varphi[\ell - 1]) = sp(s[\ell + 1],\ sp(stmt(\ell), \varphi[\ell - 1])).$$

   As before, for proper bookkeeping, all occurrences of variables $b_i[next]$ in the expression for $sp(stmt(\ell), \varphi[\ell - 1])$ are swapped with $b_i[curr]$ before propagating through $stmt[\ell + 1]$.

4. Conditional statement: `if` (g) `then` $S_{\texttt{if}}$ `else` $S_{\texttt{else}}$ `fi`.

   The strongest postcondition of a conditional statement is computed as the following disjunction:

$$\begin{aligned}\varphi[\ell] = sp(S_{\texttt{if}},\ \varphi[\ell-1] \wedge g[curr/init])\ \vee \\ sp(S_{\texttt{else}},\ \varphi[\ell-1] \wedge \neg g[curr/init]).\end{aligned}$$

   If $g = *$, the strongest postcondition is computed as the *disjunction* of the strongest postconditions over $S_{\texttt{if}}$ and $S_{\texttt{else}}$: $sp(S_{\texttt{if}}, \varphi[\ell-1]) \vee sp(S_{\texttt{else}}, \varphi[\ell-1])$.

5. Loop statement: `while` (g) `do` $S_{\texttt{loop}}$ `od`.

   Similar to postcondition propagation, precondition propagation of loop statements involves fixpoint computation over the *sp* operator. We define the strongest postcondition of the i^{th} loop iteration, $sp(S^i_{\texttt{loop}}, \varphi[\ell-1])$, as the smallest set of variable valuations in which execution of the loop body is guaranteed to terminate, in *at most* i iterations, starting in any state satisfying $\varphi[\ell-1]$. This is computed as follows:

$$\begin{aligned}sp(S^i_{\texttt{loop}}, \varphi[\ell-1]) &= \bigvee_{j=0}^{i} (Y_j \wedge \neg g), \\ \text{where,}\quad Y_0 &= \varphi[\ell-1], \\ Y_j &= sp(S_{\texttt{loop}}, Y_{i-1} \wedge g).\end{aligned}$$

   Note that, as before, each Y_j is computed in terms of variables in $V[next]$ (and $V[init]$), which are then swapped with their respective copies in

$V[curr]$ before computing $sp(S_{\texttt{loop}}, Y_{j-1} \wedge g)$. Let us suppose the fixpoint computation terminates after I iterations. The strongest postcondition for the loop statement, which corresponds to the smallest set of variable valuations such that the loop, when begun in any state satisfying $\varphi[\ell-1]$ is guaranteed to terminate in one of them, is thus given by:

$$\begin{aligned}\varphi[\ell] &= sp(S^I_{\texttt{loop}}, \varphi[\ell-1]) \\ &= \bigvee_{j=0}^{I} (Y_j \wedge \neg g) \\ &= \neg g \wedge (\varphi[\ell-1] \vee \bigvee_{j=1}^{I} sp(S_{\texttt{loop}}, Y_{j-1} \wedge g)).\end{aligned}$$

If $g = *$, the strongest postcondition is expressed as the fixpoint: $\varphi[\ell] = \bigvee_{j=0}^{I} Z_i$, where $Z_0 = \varphi[\ell-1]$ and $Z_j = sp(S_{\texttt{loop}}, Z_{j-1})$.

The strongest postcondition of a loop statement is undefined if there exists *some* variable valuation in $\varphi[\ell-1]$, starting from which there exists some non-terminating execution of the loop. Hence, the above computation for the strongest postcondition of a loop is accompanied by a check for termination for *all* variable valuations satisfying $\varphi[\ell-1]$. The termination check can be done in multiple ways. One approach is to propagate the prospective strongest postcondition computed above, back through the loop, and check if the weakest precondition so obtained contains $\varphi[\ell-1]$. If this check is `false`, it implies the existence of states in $\varphi[\ell-1]$ that do not terminate.

If the strongest postcondition $sp(stmt(\ell), \varphi[\ell-1])$ evaluates to `false` or is

undefined for any statement $stmt(\ell)$, our algorithm aborts propagation of preconditions, and proceeds to the next phase — repair generation.

3.3 Step II: Repair Generation

We now present an algorithm to repair an annotated program $\mathcal{B}$ that does not satisfy its specification ($\langle\varphi\rangle\,\mathcal{B}\,\langle\psi\rangle$ is `false`). Before proceeding, we provide an alternate, equivalent characterization of our notion of correctness in Lem. 3.3.1 and then establish an important result in Lem. 3.3.2.

Lem. 3.3.1 is stated without proof. The first part of the lemma is standard [37], and the second part of the lemma follows from our definition of strongest postconditions.

Lemma 3.3.1. *Characterization of total correctness:*
For any statement s in program $\mathcal{B}$,

$$\begin{aligned} \langle\varphi\rangle\, s\, \langle\psi\rangle &\equiv \varphi \Rightarrow wp(s,\psi), \\ \langle\varphi\rangle\, s\, \langle\psi\rangle &\equiv \begin{cases} sp(s,\varphi) \Rightarrow \psi, & \textit{when } sp(s,\varphi) \textit{ is defined,} \\ \texttt{false}, & \textit{otherwise.} \end{cases} \end{aligned}$$

As described in Sec. 3.2, we annotate each program statement $stmt(\ell)$ in $\mathcal{B}$ with a propagated precondition $\varphi[\ell-1]$ and a propagated postcondition $\psi[\ell]$. This provides us with n *local* Hoare triples, $\langle\varphi[\ell-1]\rangle\, stmt(\ell)\, \langle\psi[\ell]\rangle$, for $\ell \in [1,n]$. In the following lemma, we establish an interesting relation between the local Hoare triples and the Hoare triple for the entire program. This lemma is the basis for our repair algorithm. We claim that $\langle\varphi\rangle\,\mathcal{B}\,\langle\psi\rangle$ is `false` if and

only if all the local Hoare triples are `false`. Further, all the local Hoare triples are `false` if and only if any one local Hoare triple is `false`.

Lemma 3.3.2. *For a Boolean program $\mathcal{B}$ composed of a sequence of n statements, the following expressions are equivalent when all strongest postconditions $\psi[1], \psi[2], \ldots, \psi[n]$ are defined:*

$$\langle \varphi \rangle \, \mathcal{B} \, \langle \psi \rangle,$$
$$\exists \ell \in [1, n] : \langle \varphi[\ell - 1] \rangle \; stmt(\ell) \; \langle \psi[\ell] \rangle,$$
$$\forall \ell \in [1, n] : \langle \varphi[\ell - 1] \rangle \; stmt(\ell) \; \langle \psi[\ell] \rangle.$$

Proof. We first prove an equivalence between the second and third expressions. Let $stmt(\ell - 1)$**;** $stmt(\ell)$**;** $stmt(\ell + 1)$ be a sequence of any three consecutive statements of program $\mathcal{B}$. Consider the Hoare triple, $\langle \varphi[\ell - 1] \rangle \; stmt(\ell) \; \langle \psi[\ell] \rangle$. We have:

$\langle \varphi[\ell - 1] \rangle \; stmt(\ell) \; \langle \psi[\ell] \rangle$

$$\equiv \varphi[\ell - 1] \Rightarrow wp(stmt(\ell), \psi[\ell]) \qquad (Lem.\ 3.3.1)$$
$$\equiv sp(stmt(\ell - 1), \varphi[\ell - 2]) \Rightarrow \psi[\ell - 1] \qquad \text{(by definition)}$$
$$\equiv \langle \varphi[\ell - 2] \rangle \; stmt(\ell - 1) \; \langle \psi[\ell - 1] \rangle \qquad (Lem.\ 3.3.1).$$

Similarly, we can show $\langle \varphi[\ell - 1] \rangle \; stmt(\ell) \; \langle \psi[\ell] \rangle \equiv \langle \varphi[\ell] \rangle stmt(\ell + 1) \langle \psi[\ell + 1] \rangle$. Extending this result to program statements preceding $stmt(\ell - 1)$ and succeeding $stmt(\ell + 1)$, we arrive at an equivalence between the second and third expressions.

We prove an equivalence between the first and third expressions as follows:

$\langle\varphi\rangle\, \mathcal{B}\, \langle\psi\rangle$

$\equiv \varphi \Rightarrow wp(\mathcal{B}, \psi)$ (*Lem.* 3.3.1)

$\equiv \varphi[0] \Rightarrow wp(stmt(1); stmt(2); \ldots; stmt(n),\ \psi)$

$\equiv \varphi[0] \Rightarrow wp(stmt(1); stmt(2); \ldots; stmt(n-1),\ \psi[n-1])$

$\equiv \ldots$

$\equiv \varphi[0] \Rightarrow wp(stmt(1), \psi[1])$

$\equiv \langle\varphi[0]\rangle\ stmt(1)\ \langle\psi[1]\rangle$ (*Lem.* 3.3.1).

A similar equivalence can be obtained between $\langle\varphi\rangle\, \mathcal{B}\, \langle\psi\rangle$, and every local Hoare triple $\langle\varphi[\ell-1]\rangle\ stmt(\ell)\ \langle\psi[\ell]\rangle$, $\ell \in [2, n]$.

□

3.3.1 The Repair Algorithm

A consequence of Lemma 3.3.2 is that if the ℓ^{th} Hoare triple is made `true` by repairing $stmt(\ell)$, then the program so obtained satisfies the specification. Hence, our repair strategy proceeds by examining program statements in some specific order to identify potential candidates for repair. If propagation of postconditions is aborted due to an empty weakest precondition for statement $stmt(\ell)$, we check for repairable statements among the statements $stmt(\ell), stmt(\ell+1), \ldots, stmt(n)$. Similarly, if propagation of preconditions is aborted due to an undefined or empty strongest postcondition for

statement $stmt(\ell)$, we check for repairable statements among the statements $stmt(0), stmt(1), \ldots, stmt(\ell)$.

Recall that the set of permissible update schemas is:

$$\mathcal{U} = \{id, \texttt{assign} \mapsto \texttt{skip}, \texttt{cond} \mapsto \texttt{skip}, \texttt{loop} \mapsto \texttt{skip},$$
$$\texttt{assign} \mapsto \texttt{assign}, \texttt{if} \mapsto \texttt{if}, \texttt{while} \mapsto \texttt{while}\}.$$

A single application of an update schema from $\mathcal{U}$ is thus a deletion or a specific modification of an assignment, conditional or loop statement. We assume that we have a set of statements (by default, the set of all n statements) of $\mathcal{B}$ to be inspected for repairability, in some predecided order. For every statement $stmt(\ell)$ to be inspected, we pose a query to check if $stmt(\ell)$ can be repaired by a single application of an update schema from $\mathcal{U}$. If the query returns `true` for any statement, we proceed to synthesize a repair and declare success. If not, we proceed to the next statement in the given order. If none of the statements can be repaired, we report failure in repairing the program given the current constraints. In what follows, we explain how to formulate `Query` for each permissible update schema and synthesize the corresponding repair:

Statement Deletion. Before checking if $stmt(\ell)$ can be repaired by modifying it, we first do a simple check to see if it can be repaired by deleting it. We can do this by formulating `Query` as the following quantified Boolean formula (QBF):

$$\forall b[init] \in V[init] : \ \varphi[\ell - 1] \Rightarrow \psi[\ell][\forall i \in [1, r] : b_i[curr]/b_i[next]]. \qquad (3.1)$$

This QBF checks if the local Hoare triple given by $\langle \varphi[\ell-1] \rangle\, stmt(\ell)\, \langle \psi[\ell] \rangle$ can be made `true` if $stmt(\ell)$ is deleted. If this QBF is `true`, we return a new repaired program $\widehat{\mathcal{B}}$ with $stmt(\ell)$ replaced by `skip`. If this QBF is `false`, we proceed to check if $stmt(\ell)$ can be repaired by modifying it.

Assignment Statement Modification. Suppose $stmt(\ell)$ is an assignment statement, $b_1, \ldots, b_m := e_1, \ldots, e_m$. Let $\widehat{stmt}(\ell) : b_1, \ldots, b_m := \widehat{e}_1, \ldots, \widehat{e}_m$ denote a potential repair for $stmt(\ell)$, assigning expression $\widehat{e}_i$ to variable b_i for each $i \in [1, m]$. We check for the existence of a suitable $\widehat{stmt}(\ell)$ by formulating `Query` as the following QBF:

$$\forall b[init] \in V[init]\ \exists z_1, \ldots, z_m :\ \varphi[\ell-1] \Rightarrow \widehat{\psi}[\ell-1], \tag{3.2}$$

where z_i represents a valuation of the unknown expression $\widehat{e}_i$ for each $i \in [1, m]$ and $\widehat{\psi}[\ell-1] = wp(b_1, \ldots, b_m := z_1, \ldots, z_m, \psi[\ell])$. Thus, `Query` poses the question: does there exist a valuation z_i of variable b_i, for $i \in [1, m]$, which makes the local Hoare triple $\langle \varphi[\ell-1] \rangle\ b_1, \ldots, b_m := z_1, \ldots, z_m\ \langle \psi[\ell] \rangle$ `true`.

If the above QBF evaluates to `true`, the next key step is to obtain expressions for $\widehat{e}_1, \ldots, \widehat{e}_m$ in terms of variables in $V[curr]$. In general, one can derive such expressions from the certificates of validity generated by a QBF solver. In what follows, we outline a way to compute a solution for the expressions without using a QBF solver.

Let us first derive our solution for an assignment to a single variable; thus, $stmt(\ell)$ is the statement $b := e$, $\widehat{stmt}(\ell)$ is the statement $b := \widehat{e}$ and z

represents a valuation of $\widehat{e}$. Let us denote the expression $\varphi[\ell-1] \Rightarrow \widehat{\psi}[\ell-1]$ by T. Let $T|_z$, $T|_{\neg z}$ denote the positive, negative cofactors of T w.r.t. z, i.e. the expressions obtained by substituting all occurrences of z in T by `true`, `false`, respectively. When the QBF in (3.2) evaluates to `true`, for each initial valuation of the program variables, $\exists z : T$ is `true`, i.e., T is `true` for $z =$ `true` or for $z =$ `false`. Thus, (a) T is `true` and $z =$ `true`, in which case, $T = T|_z =$ `true` (and the value of $T|_{\neg z}$ doesn't matter), or, (b) T is `true` and $z =$ `false`, in which case, $T = T|_{\neg z} =$ `true` (and $T|_z$ doesn't matter). Let us set $T|_{\neg z} =$ `false` in case (a) and set $T|_z =$ `true` in case (b). This fixes two possible solutions (of many): either $z = T|_z$ or $z = \neg T|_{\neg z}$ can serve as a witness to the validity of the QBF in (3.2) and could yield $\widehat{e}$.

One can extend the above solution to parallel assignments using similar reasoning. For instance, for assignments to 2 variables, $z_1 = T|_{z_1 z_2} \vee T|_{z_1 \neg z_2}$ and $z_2 = T|_{z_1 z_2} \vee T|_{\neg z_1 z_2}$ form a witness to the validity of the QBF in (3.2) and could yield $\widehat{e}_1$, $\widehat{e}_2$, respectively.

Note, however, that the expressions $T|_z$, $T|_{z_1 z_2}$ etc. contain variables from both $V[init]$ and $V[curr]$. Hence the repair algorithm tries to express variables in $V[init]$ in terms of variables in $V[curr]$. If this is feasible, the repaired expressions, $\widehat{e}_1, \ldots, \widehat{e}_m$, are obtained solely in terms of variables from $V[curr]$. If this cannot be done, the repair algorithm suggests adding at most m new constants, $binit_1, binit_2, \ldots, binit_m$, to store the initial values of the program variables. Note that this is the only time the repair algorithm suggests addition of constants or insertion of statements. The following lemma states

the soundness of the above repair generation.

Lemma 3.3.3. *When the QBF in (3.2) is* `true`*, replacing the statement* $stmt(\ell)$*:* $b_1, \ldots, b_m := e_1, \ldots, e_m$ *by* $\widehat{stmt}(\ell)$*:* $b_1, \ldots, b_m := \widehat{e}_1, \ldots, \widehat{e}_m$*, with* $\widehat{e}_1, \ldots, \widehat{e}_m$ *obtained as above, makes the Hoare triple* $\langle \varphi[\ell-1] \rangle\, \widehat{stmt}(\ell)\, \langle \psi[\ell] \rangle$ `true`*.*

Conditional Statement Modification. Suppose $stmt(\ell)$ is a conditional statement, `if` (g) `then` $S_{\texttt{if}}$ `else` $S_{\texttt{else}}$ `fi`. `Query` for $stmt(\ell)$ checks for the possibility of repairing one of three options - the guard, a statement in $S_{\texttt{if}}$, or a statement in $S_{\texttt{else}}$. Note that repairing the guard corresponds to an application of the update schema `if` $\mapsto$ `if`, while repairing a statement in $S_{\texttt{if}}$ or $S_{\texttt{else}}$ may involve any update schema in $\mathcal{U}$ that may be applied to the statements in $S_{\texttt{if}}$, $S_{\texttt{else}}$.

Let $\widehat{stmt}(\ell)$: `if` $(\widehat{g})$ `then` $S_{\texttt{if}}$ `else` $S_{\texttt{else}}$ `fi` denote a potential repair that modifies the guard in $stmt(\ell)$. We check for the existence of such a $\widehat{stmt}(\ell)$ by formulating a QBF similar to (3.2):

$$\forall b[init] \in V[init]\, \exists z :\ \varphi[\ell-1] \Rightarrow \widehat{\psi}[\ell-1], \tag{3.3}$$

where z represents a valuation of the unknown expression $\widehat{g}$, and $\widehat{\psi}[\ell-1] = wp(\texttt{if}\,(z)\,\texttt{then}\,S_{\texttt{if}}\,\texttt{else}\,S_{\texttt{else}}\,\texttt{fi},\ \psi[\ell])$. If the QBF is `true`, we can derive $\widehat{g}$ as before from, say, the positive cofactor of the expression $\varphi[\ell-1] \Rightarrow \widehat{\psi}[\ell-1]$. If the QBF is not `true`, we make two separate sets of queries to check if a statement in $S_{\texttt{if}}$ or in $S_{\texttt{else}}$ can be repaired using some permissible update

schema to make the corresponding local Hoare triples `true`. Thus, for every statement *stmt* in $S_{\texttt{if}}$ and every potential repair $\widehat{stmt}$ corresponding to an applicable update schema from $\mathcal{U}$, we formulate a similar QBF; if any of the QBFs is `true`, we compute $\widehat{stmt}$, thereby yielding a modified $\widehat{S}_{\texttt{if}}$. If none of the QBFs is `true`, we repeat the process for every statement in $S_{\texttt{else}}$ and attempt to compute a modified $\widehat{S}_{\texttt{else}}$ that makes the local Hoare triple `true`. The following lemma states the soundness of the above repair generation.

Lemma 3.3.4. *When the QBF in (3.3) is* `true`*, replacing the conditional statement* $stmt(\ell)$*:* `if` (g) `then` $S_{\texttt{if}}$ `else` $S_{\texttt{else}}$ `fi` *by the statement* $\widehat{stmt}(\ell)$*:* `if` $(\widehat{g})$ `then` $S_{\texttt{if}}$ `else` $S_{\texttt{else}}$ `fi`*, with* $\widehat{g}$ *obtained as above, makes the Hoare triple* $\langle\varphi[\ell-1]\rangle\ \widehat{stmt}(\ell)\ \langle\psi[\ell]\rangle$ `true`*. When the query for repairing a statement in* $S_{\texttt{if}}$*,* $S_{\texttt{else}}$ *is* `true`*, replacing* $stmt(\ell)$ *by* $\widehat{stmt}(\ell)$*:* `if` (g) `then` $\widehat{S}_{\texttt{if}}$ `else` $S_{\texttt{else}}$ `fi`*,* $\widehat{stmt}(\ell)$*:* `if` (g) `then` $S_{\texttt{if}}$ `else` $\widehat{S}_{\texttt{else}}$ `fi`*, respectively makes the Hoare triple* $\langle\varphi[\ell-1]\rangle\ \widehat{stmt}(\ell)\ \langle\psi[\ell]\rangle$ `true`*.*

Loop Statement Modification. Suppose $stmt(\ell)$ is a loop statement, `while` (g) `do` $S_{\texttt{loop}}$ `od`. `Query` for a loop statement checks for the possibility of repairing either the loop guard or a statement in $S_{\texttt{loop}}$. `Query` and repair for loop statements have an additional responsibility of ensuring termination of the repaired loop, and hence, differ from the versions we have seen so far.

Let us first focus on repairing the loop guard. Recall that before checking if $stmt(\ell)$ can be repaired by modifying it, we first check if it can be repaired by deleting it. Hence, in what follows, we can assume that the loop

statement cannot be repaired by deleting it. Thus, some states satisfying the precondition $\varphi[\ell-1]$ must enter the loop, i.e., any repair $\widehat{g}$ for the loop guard must include some states satisfying $\varphi[\ell-1]$. Consider the terminating fixpoint over the monotonically decreasing sets, $\bigwedge_j Y_j$, where:

$$Y_0 = \varphi[\ell-1] \wedge \neg\psi[\ell],$$
$$Y_j = sp(S_{\texttt{loop}}, Y_{j-1}) \wedge \neg\psi[\ell].$$

This fixpoint represents the set of all states that satisfy the precondition $\varphi[\ell-1]$, but not the postcondition $\psi[\ell]$, enter the loop, execute it iteratively and never satisfy the postcondition $\psi[\ell]$ on completion of a loop execution. Suppose this fixpoint does not evaluate to `false`. On the one hand, the loop guard $\widehat{g}$ that can repair the loop statement must include the states satisfying this non-empty fixpoint, as these states do not satisfy $\psi[\ell]$ on exiting the loop. On the other hand, $\widehat{g}$ cannot include the set of states satisfying this fixpoint as such states will never transition into states that can exit the loop while satisfying $\psi[\ell]$. Thus, the states from this set cannot belong to either $\widehat{g}$ or $\neg\widehat{g}$, leading to a contradiction.

Hence, if a loop statement cannot be repaired by deleting it, a choice for a loop guard that can help repair the loop statement exists if and only if the above fixpoint is empty.

Assuming that the above fixpoint is empty, consider the fixpoint over the monotonically increasing sets $\bigvee_j Y_j$, where each Y_j is defined the same as above. By construction, this fixpoint accumulates all states that satisfy

the precondition, enter the loop, but do not satisfy the postcondition $\psi[\ell]$ on completion of a loop execution. If we select the loop guard to be this fixpoint, then any state that exits the loop is forced to satisfy $\psi[\ell]$ by construction. Thus each iterant of this fixpoint accumulates states that are chosen by the guard to execute the loop during that iteration. Moreover, since the fixpoint over $\bigwedge_j Y_j$ is known to be empty, we can be assured that there are no states in $\widehat{g}$ that belong to a non-terminating execution. Note that g is not `false` as the first iterant $Y_0 = \varphi[\ell - 1] \wedge \neg\psi[\ell]$ is required to be non-empty. If Y_0 were empty or `false`, the QBF in (3.1) is `true` as $\varphi[\ell - 1] \Rightarrow \psi[\ell]$. Hence, the loop statement could have been repaired by deleting it, thereby contradicting our earlier assumption. The soundness of our repair choice for the loop guard is stated in the following lemma.

Lemma 3.3.5. *Let* $Y_0 = \varphi[\ell - 1] \wedge \neg\psi[\ell]$ *and* $Y_j = sp(S_{\texttt{loop}}, Y_{j-1}) \wedge \neg\psi[\ell]$. *When* $stmt(\ell)$*:* `while` (g) `do` $S_{\texttt{loop}}$ `od` *cannot be repaired by deleting it, replacing* $stmt(\ell)$ *with* $\widehat{stmt}(\ell)$*:* `while` $(\widehat{g})$ `do` $S_{\texttt{loop}}$ `od`*, where* $\widehat{g}$ *is the fixpoint over the sets* $\bigvee_j Y_j$*, makes the Hoare triple* $\langle\varphi[\ell - 1]\rangle\ \widehat{stmt}(\ell)\ \langle\psi[\ell]\rangle$ `true` *if and only if the fixpoint* $\bigwedge_j Y_j$ *evaluates to* `false`*.*

If the fixpoint over $\bigwedge_j Y_j$ is not `false`, the loop guard cannot be repaired; we then formulate a set of queries to check if any statement in $S_{\texttt{loop}}$ can be repaired. To keep the exposition simple, we show how to formulate a query for checking if an assignment statement in $S_{\texttt{loop}}$ can be repaired. The method extends inductively to inner conditional and loop statements. Let $stmt(p)$ be

an assignment statement in $S_{\texttt{loop}}$. As before, let $\widehat{stmt}(p)$ denote a potential repair for $stmt(\ell)$. Let $\widehat{S}_{\texttt{loop}}$ denote the updated loop body. `Query` returns `true` if the following QBF is `true`, and `false` otherwise:

$$\forall b[init] \in V[init]\, \exists z_1, \ldots, z_m :\ \varphi[\ell - 1] \Rightarrow \widehat{\psi}[\ell - 1],$$

where each z_i is as before and $\widehat{\psi}[\ell-1]$ is the weakest precondition of $\widehat{S}_{\texttt{loop}}$ with respect to $\psi[\ell]$. To compute this weakest precondition, we compute a fixpoint as outlined in Sec. 3.2.1, noting that each iterant of the fixpoint, and hence, the final fixpoint would be an expression over the variables $z_1, \ldots, z_m$, $V[init]$ and $V[curr]$. If the QBF is `true`, we can derive $z_1, \ldots, z_m$ from the cofactors of $\varphi[\ell - 1] \Rightarrow \widehat{\psi}[\ell - 1]$ as before. Since the weakest precondition computation guarantees termination, the repaired loop is guaranteed to terminate. If the QBF is not `true`, we attempt to repair another statement within the loop body. The soundness of this repair strategy is stated in the following lemma.

Lemma 3.3.6. *If* `Query` *for repairing statement* $stmt(p)$ *in the loop body of* $stmt(\ell)$*:* `while` (g) `do` $S_{\texttt{loop}}$ `od` *is* `true`*, then replacing* $stmt(\ell)$ *by* $\widehat{stmt}(\ell)$*:* `while` (g) `do` $\widehat{S}_{\texttt{loop}}$ `od`*, where* $\widehat{S}_{\texttt{loop}}$ *is the updated loop body with* $stmt(p)$ *replaced with* $\widehat{stmt}(p)$ *and* $\widehat{stmt}(p)$ *is computed as outlined above, makes the Hoare triple* $\langle \varphi[\ell - 1] \rangle\, \widehat{stmt}(\ell)\, \langle \psi[\ell] \rangle$ `true`*.*

Example: In the annotated program from Fig. 3.2, the QBF for the 2^{nd} statement evaluates to `true`, and generates the expected repair: $b_2[next] := b_1[curr] \oplus b_2[curr]$; for the program.

3.3.2 Algorithm Notes

Correctness. The following theorem states that our algorithm is sound and complete with respect to our repair constraints.

Theorem 3.3.7. *Given a precondition, postcondition pair (φ, ψ), a Boolean program $\mathcal{B}$ such that $\langle\varphi\rangle\,\mathcal{B}\,\langle\psi\rangle$ is not valid, and the set of permissible, statement-level update schemas given by:*

$$\mathcal{U} = \{id, \texttt{assign} \mapsto \texttt{skip}, \texttt{cond} \mapsto \texttt{skip}, \texttt{loop} \mapsto \texttt{skip},$$
$$\texttt{assign} \mapsto \texttt{assign}, \texttt{if} \mapsto \texttt{if}, \texttt{while} \mapsto \texttt{while}\},$$

if there exists a Boolean program $\widehat{\mathcal{B}}$ such that $\langle\varphi\rangle\,\widehat{\mathcal{B}}\,\langle\psi\rangle$ is valid, and $\widehat{\mathcal{B}}$ can be obtained by a single *application of an update schema from $\mathcal{U}$, our algorithm finds one such $\widehat{\mathcal{B}}$. If our algorithm finds a $\widehat{\mathcal{B}}$, then $\langle\varphi\rangle\,\widehat{\mathcal{B}}\,\langle\psi\rangle$ is guaranteed to be valid and $\widehat{\mathcal{B}}$ is guaranteed to differ from $\mathcal{B}$ in exactly one statement, with the statement modification corresponding to an application of an update schema from $\mathcal{U}$.*

Proof. The soundness result follows from Lem. 3.3.1, Lem. 3.3.2, Lem. 3.3.3, Lem. 3.3.4, Lem. 3.3.5 and Lem. 3.3.6. The completeness result is a direct consequence of our query formulation, which checks for the existence of exactly one statement modification corresponding to an application of an update schema from $\mathcal{U}$. □

We emphasize that by completeness, we refer to the completeness of our algorithm for repair of Boolean programs. When the algorithm is used for

predicate abstraction-based repair of concrete programs, the overall algorithm can no longer be complete. Given the repair constraints, it may be possible to repair a concrete program $\mathcal{P}$, but it may not be possible to repair the corresponding Boolean program $\mathcal{B}$ to eliminate all its spurious counterexample paths.

Complexity Analysis. All fixpoint computations in our algorithm can be done in time exponential in the number of program variables in the worst-case. Moreover, the `Query` function for assignment and conditional statements involves checking validity of a QBF with exactly two alternating quantifiers and lies in the second polynomial hierarchy, i.e., it is Σ_2^P -complete in the number of program variables. All other operations like swapping of variables, substitution, Boolean manipulation and cofactor computation can be done in either constant time or time exponential in the number of program variables in the worst-case.

Thus, the worst-case complexity of our algorithm is exponential in the number of program variables. In practice, most of these computations can be done very efficiently using BDDs. The initial results obtained from a preliminary implementation of our algorithm that uses a Java-based BDD library [131] have been promising. Use of BDDs allows symbolic storage and efficient manipulation of preconditions and postconditions as well as efficient computation of fixpoints. The `forall` and `exist` methods for BDDs facilitate computing validity of a desired QBF and hence enable the `Query` check. Similarly, the

repair operation can be performed easily by computing the requisite cofactor of a BDD.

3.3.3 Annotation and Repair of Programs with Procedures

Our algorithm can be extended to programs containing non-recursive procedure calls. The basic idea is to use *procedure summaries* to characterize the effect of a procedure. Let F be a procedure in program $\mathcal{B}$, with a set $FV = \{\beta_1, \ldots, \beta_k\}$ of formal parameters, a set LV of local variables not included in FV, and a sequence S of statements. Let $GV[before]$ and $GV[after]$ be two copies of the set GV of global variables to represent the global variable valuations before and after execution of F, respectively. Let $GV[curr]$ and $GV[next]$ be two copies of GV, as used earlier in this chapter, to aid propagation within the procedure

Annotation. The *backward summary* of F, denoted Δ_F^-, is an expression over variables in FV, $GV[before]$ and $GV[after]$. Δ_F^- can be computed by eliminating all local variables in LV from the weakest precondition of the predicate $\bigwedge_{b_i \in GV} b_i[before] = b_i[after]$, over the procedure body S [9]. The *forward summary* of F, denoted Δ_F^+, is also an expression over FV, $GV[before]$ and $GV[after]$. Δ_F^+ can be computed from the strongest postcondition of the predicate $\bigwedge_{b_i \in GV} b_i[curr] = b_i[before]$, over the procedure body S, based on the rules described in Sec. 3.2.2, to first yield an expression over variables in $GV[next]$, $GV[before]$, FV and LV. Δ_F^+ is then obtained by replacing the variables in $GV[next]$ with their copies in $GV[after]$ and eliminating all local

variables in LV.

Δ_F^-, Δ_F^+ are used at each call-site of F during postcondition, precondition propagation, respectively. Let $stmt(\ell)$: `call` $F(\alpha_1, \ldots, \alpha_k)$. To propagate of $\psi[\ell]$ - an expression over $V[next]$ and $V[init]$ - back through $stmt(\ell)$ to compute $\psi[\ell - 1]$ - an expression over $V[curr]$ and $V[init]$, the global variable valuations in $\psi[\ell]$ are matched up with those after execution of F, all global variables in $GV[before]$ in Δ_F^- are replaced with their copies in $GV[curr]$, and the formal parameters of F are replaced with the corresponding actual parameters. To propagate of $\varphi[\ell - 1]$ - an expression over $V[curr]$ and $V[init]$ - forward through $stmt(\ell)$ to compute $\varphi[\ell]$ - an expression $V[next]$ and $V[init]$, we essentially do a conjunction of Δ_F^+ and $\varphi[\ell - 1]$, taking care to match up appropriate copies of global variables, and replace all formal parameters of F with the corresponding actual parameters.

Since we preclude recursive calls, the call-graph for our programs is acyclic, and hence can be sorted in reverse topological order. We compute procedure summaries in this order to ensure that the summaries are always available at the call-site of each function.

Repair. To repair a statement within procedure F, we proceed as before, first attempting to replace the statement by `skip`, failing which, attempting to modify the statement by replacing expressions with fresh variables (z_1, z_2 etc.). Note however that once a procedure is considered for repair, the procedure summaries need to be recomputed and the program needs to be re-annotated, before we can solve for the fresh variables.

Chapter 4

Cost-Aware Program Repair

Overview. In this chapter, we propose *cost-aware* repair of programs annotated with assertions. We present an algorithm that can repair erroneous Boolean programs by modifying them in multiple locations such that the total modification cost does not exceed a specified repair budget; our algorithm for Boolean program repair is sound and complete for partial correctness. We also describe strategies for concretization of repaired Boolean programs to enable predicate abstraction-based repair of concrete programs. We demonstrate the efficacy of our overall framework by repairing C programs using a prototype tool.

In Chapter 3, we focused on repairing a class of Boolean programs using a single application of a permissible update schema. In this chapter, we generalize the earlier method in several ways. We present a predicate abstraction-based framework for repairing infinite-state programs, annotated with multiple assertions, and with possibly recursive procedures, with respect to partial correctness[1]. Along with a repaired program, the framework also generates a

[1]Our approach can be extended to handle total correctness as well.

proof of correctness consisting of *inductive assertions*. The methodology presented in this chapter can repair programs by modifying them in multiple program locations, and can ensure the readability of the repaired program using user-defined expression templates. Last, but not the least, the framework is cost-aware - given a user-defined cost function, that charges each application of an update schema to a program statement some user-defined cost, and a repair budget, the framework computes a repaired program whose total modification cost does not exceed the repair budget; we postulate that this cost-aware formulation is a flexible and convenient way of incorporating expert programmer intent and intuition in automated program debugging.

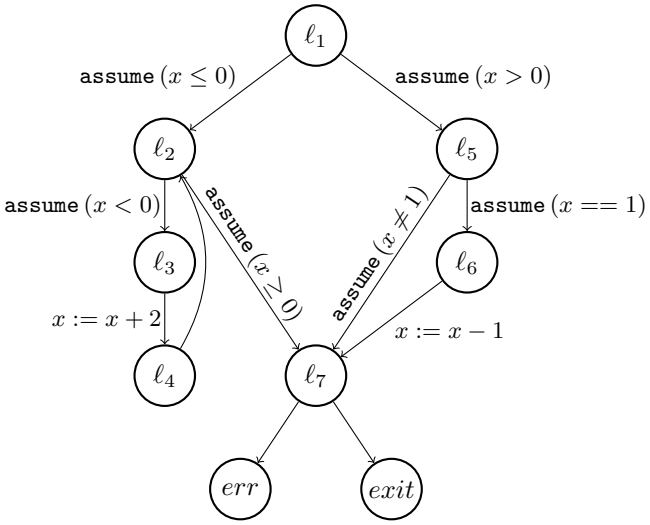
4.1 Formal Framework

Recall the example from Chapter 2, shown here in Fig. 4.1. In this chapter, we will use the transition graph representation of programs. As can be seen from Fig. 4.1b and Fig. 4.1d, it suffices to consider the set of statement types given by: $\Sigma = \{\texttt{skip}, \texttt{assign}, \texttt{assume}, \texttt{assert}, \texttt{call}, \texttt{return}, \texttt{goto}\}$. Observe that the Boolean expressions in the `assume` statements labeling the edges in the transition graphs are always deterministic. Thus, a concrete program contains no nondeterministic expressions, and a Boolean program contains nondeterministic expressions only in the RHS of assignment statements[2].

[2]We remind the reader that our prototype tool (see Sec. 4.4) can handle arbitrary nondeterministic Boolean expressions, and not just $*$ expressions. Also, our tool accepts Boolean programs with `if` statements (the tool we use for predicate abstraction encodes all `while` statements using conditional and `goto` statements).

```
main() {
   int x;
   ℓ1 : if (x ≤ 0)
   ℓ2 :    while (x < 0){
   ℓ3 :      x := x + 2;
   ℓ4 :      skip;
           }
        else
   ℓ5 :    if (x == 1)
   ℓ6 :      x := x − 1;
   ℓ7 : assert (x > 1);
}
```

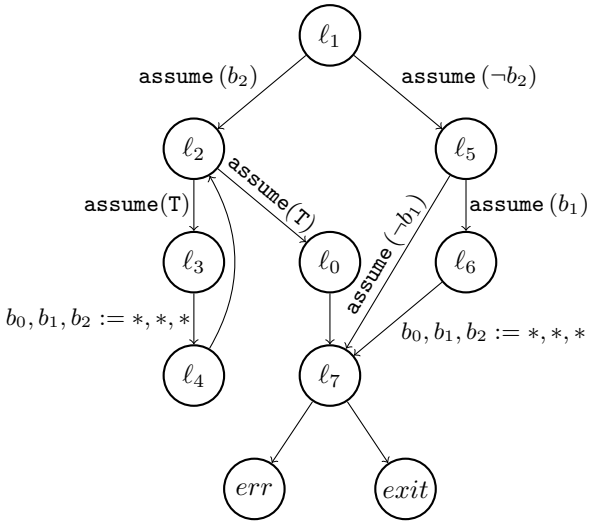
(a) $\mathcal{P}$



(b) $\mathcal{G}(\mathcal{P})$

```
main() {
   /∗ γ(b0) = x ≤ 1 ∗/
   /∗ γ(b1) = x == 1 ∗/
   /∗ γ(b2) = x ≤ 0 ∗/
   Bool b0, b1, b2 := ∗, ∗, ∗;
   ℓ1 : if (¬b2) then goto ℓ5;
   ℓ2 : if (∗) then goto ℓ0;
   ℓ3 : b0, b1, b2 := ∗, ∗, ∗;
   ℓ4 : goto ℓ1;
   ℓ0 : goto ℓ7;
   ℓ5 : if (¬b1) then goto ℓ7;
   ℓ6 : b0, b1, b2 := ∗, ∗, ∗;
   ℓ7 : assert (¬b0);
}
```

(c) $\mathcal{B}$



(d) $\mathcal{G}(\mathcal{B})$

Figure 4.1: An example concrete program $\mathcal{P}$, a corresponding Boolean program $\mathcal{B}$ and their transition graphs

We assume that program correctness is specified using a set of `assert` statements of the form ℓ : `assert` (g), included in the program. Recall that a program annotated with a set of assertions is partially correct iff every finite execution path of $\mathcal{P}$ ends in a terminating configuration. The program is totally correct iff every execution path is finite and ends in a terminating configuration. While the approach presented in this chapter can be extended to handle total correctness, we only present a treatment of partial correctness here; henceforth, correctness and partial correctness are used interchangeably.

In this chapter, we fix the following set of permissible update schemas for programs:

$$\mathcal{U} = \{id, \texttt{assign} \mapsto \texttt{assign}, \texttt{assign} \mapsto \texttt{skip}, \texttt{assume} \mapsto \texttt{assume}, \\ \texttt{call} \mapsto \texttt{call}, \texttt{call} \mapsto \texttt{skip}\}.$$

Let $c_{\mathcal{U},\mathcal{L}} : \mathcal{U} \times \mathcal{L} \to \mathbb{N}$ be a cost function that maps a tuple, consisting of a statement-level update schema u and a location ℓ of $\mathcal{P}$, to a certain cost. Thus, $c_{\mathcal{U},\mathcal{L}}(u, \ell)$ is the cost of applying update schema u to $stmt(\ell)$. We impose an obvious restriction on $c_{\mathcal{U},\mathcal{L}}$: $\forall \ell \in \mathcal{L} : c_{\mathcal{U},\mathcal{L}}(id, \ell) = 0$. Recall that $\mathbb{R}_{\mathcal{U},\mathcal{L}} : \mathcal{L} \to \mathcal{U}$ denotes an update function that maps each location of $\mathcal{P}$ to an update schema in $\mathcal{U}$. Since we have already fixed the set $\mathcal{U}$ and the set $\mathcal{L}$ of locations of program $\mathcal{P}$ (or equivalently, of Boolean program $\mathcal{B}$[3]), we

[3]In general, as can be seen from Fig. 4.1, $\mathcal{L}(\mathcal{B})$ can be a superset of $\mathcal{L}(\mathcal{P})$. However, the extraneous locations in $\mathcal{L}(\mathcal{B})$ contain `skip` or `goto` statements and are hence, irrelevant for repair in our work.

henceforth use c, $\mathbb{R}$ instead of $c_{\mathcal{U},\mathcal{L}}$, $\mathbb{R}_{\mathcal{U},\mathcal{L}}$, respectively, The total cost, $Cost_c(\mathbb{R})$, of performing an $\mathbb{R}$-update of $\mathcal{P}$ is given by $\sum_{\ell \in \mathcal{L}} c(\mathbb{R}(\ell), \ell)$.

For this chapter, we adapt the problem definition of Sec. 2.3 as follows. Given an incorrect concrete program $\mathcal{P}$ annotated with assertions, a set of permissible, statement-level update schemas $\mathcal{U}$, a cost function c and a repair budget δ, the goal of cost-aware program repair is to compute $\widehat{\mathcal{P}}$ such that:

1. $\widehat{\mathcal{P}}$ is correct, and,
2. there exists $\mathbb{R}$:
   (a) $\widehat{\mathcal{P}}$ is some $\mathbb{R}$-update of $\mathcal{P}$, and
   (b) $Cost_c(\mathbb{R}) \leq \delta$.

In addition to the above problem, we propose another problem as follows. Let $\mathcal{T} = \{\mathcal{T}_1, \ldots, \mathcal{T}_h\}$ be a set of *templates* or *grammars*, each representing a syntactical restriction for the modified expressions in $\widehat{\mathcal{P}}$. The syntax of an example template, say $\mathcal{T}_1$, defining Boolean-valued linear arithmetic expressions over the program variables, denoted $\langle blaexpr \rangle$, is shown below:

$$
\begin{array}{lcl}
\langle blaexpr \rangle & ::= & atom \mid (\, \langle blaexpr \rangle \,) \mid \neg \langle blaexpr \rangle \mid \langle blaexpr \rangle \wedge \langle blaexpr \rangle \\
\langle atom \rangle & ::= & \langle laterm \rangle \, \langle cmp \rangle \, \langle laterm \rangle \\
\langle laterm \rangle & ::= & const \mid var \mid const \times var \mid \langle laterm \rangle + \langle laterm \rangle \\
\langle cmp \rangle & ::= & = \mid < \mid \leq .
\end{array}
$$

In the above, *const* and *var* denote integer-valued or real-valued constants and program variables, respectively. Expressions such as $v_1 + 2 \times v_2 \leq v_3$,

$(v_1 < v_2) \wedge (v_3 = 3)$ etc., that satisfy the syntactical requirements of the template $\mathcal{T}_1$, are said to belong to the *language* of the template, denoted $L(\mathcal{T}_1)$.

Let $\mathbb{E}_{\mathcal{T},\mathcal{L}} : \mathcal{L} \to \mathcal{T}$, be a function that maps each location of $\mathcal{P}$ to a template in $\mathcal{T}$. Let $\mathcal{E}(stmt(\ell))$ denote a set that includes all expressions in certain statement types and be defined as follows: if $stmt(\ell)$ is $v_1, \ldots, v_m := e_1, \ldots, e_m$, $\mathcal{E}(stmt(\ell)) = \{e_1, \ldots, e_m\}$, else if $stmt(\ell)$ is `call` $F_j(e_1, \ldots, e_k)$, $\mathcal{E}(stmt(\ell)) = \{e_1, \ldots, e_k\}$, else if $stmt(\ell)$ is `assume` (g), $\mathcal{E}(stmt(\ell)) = \{g\}$ else, $\mathcal{E}(stmt(\ell))$ is the empty set.

Given $\mathbb{E}_{\mathcal{T},\mathcal{L}}$, along with (incorrect) $\mathcal{P}$, $\mathcal{U}$, c and δ, the goal of *template-based*, cost-aware program repair is to compute $\widehat{\mathcal{P}}$ such that:

1. $\widehat{\mathcal{P}}$ is correct, and,
2. there exists $\mathbb{R}$:
   (a) $\widehat{\mathcal{P}}$ is some $\mathbb{R}$-update of $\mathcal{P}$,
   (b) $Cost_c(\mathbb{R}) \leq \delta$, and
   (c) for each location ℓ:
   $\mathbb{R}(\ell) \neq id \;\Rightarrow \forall e \in \mathcal{E}(\widehat{stmt}(\ell)) : \; e \in L(\mathbb{E}_{\mathcal{T},\mathcal{L}}(\ell))$.

We have already emphasized the benefits of formulating the program repair problem with respect to a set $\mathcal{U}$ of update schemas in Sec. 2.3. We conjecture that an insightful choice for the cost function c can further help prune the search space for repaired programs and help incorporate expert

user intuition and intent in automatic program repair. Exploration of suitable cost-functions is beyond the scope of this dissertation. For now, we would only like to emphasize that our cost-function is quite flexible, and can be used to constrain the computation of $\widehat{\mathcal{P}}$ in diverse ways. For example, the user can choose to search for $\widehat{\mathcal{P}}$ that differs from $\mathcal{P}$ in at most δ statements by defining c as:

$$\forall \ell \in \mathcal{L}, u \in \mathcal{U} : u \neq id \;\Rightarrow\; c((u, \ell)) = 1.$$

Or, the user can choose to search for $\widehat{\mathcal{P}}$ that does not modify any statement within a *trusted* procedure F_i by defining c as:

$$\forall \ell \in \mathcal{L}, u \in \mathcal{U} : u \neq id \wedge \ell \in \mathcal{L}_i \;\Rightarrow\; c((u, \ell)) = N \text{ and}$$
$$u \neq id \wedge \ell \notin \mathcal{L}_i \;\Rightarrow\; c((u, \ell)) = 1,$$

where N is some prohibitively large number. Or, the user can choose to favor the application of a particular update schema, say u_1, over others by defining c as:

$$\forall \ell \in \mathcal{L}, u \in \mathcal{U} : u \neq id \text{ and } u \neq u_1 \;\Rightarrow\; c((u, \ell)) = N \text{ and}$$
$$u = u_1 \;\Rightarrow\; c((u, \ell)) = 1,$$

where N is some prohibitively large number, and so on. Similarly, insightful templates choices can help guide the search for repairs based on user input.

As outlined in Sec. 2.4, our solution to the above program repair problems is based on predicate abstraction. Thus, in addition to the above inputs,

our framework requires (a) a Boolean program $\mathcal{B}$ such that $\mathcal{B}$ is obtained from $\mathcal{P}$ via iterative predicate abstraction-refinement and $\mathcal{B}$ exhibits a non-spurious counterexample path, and (b) the corresponding function γ that maps Boolean variables to their respective predicates. The computation of a suitable repaired program $\widehat{\mathcal{P}}$ involves two main steps:

1. Cost-aware repair of $\mathcal{B}$ to obtain $\widehat{\mathcal{B}}$, and
2. Concretization of $\widehat{\mathcal{B}}$ to obtain $\widehat{\mathcal{P}}$.

Recall that a Boolean program $\mathcal{B}$, annotated with a set of assertions, is (partially) correct iff every finite execution path of $\mathcal{B}$ ends in a terminating configuration for *all* nondeterministic choices that the program might make. The problem of cost-aware repair of a Boolean program $\mathcal{B}$ can be defined in a manner identical to cost-aware repair of a concrete program $\mathcal{P}$. Concretization of $\widehat{\mathcal{B}}$ involves mapping each statement of $\widehat{\mathcal{B}}$ that has been modified by $\mathbb{R}_{\mathcal{U},\mathcal{L}}$ into a corresponding statement of $\mathcal{P}$, using the function γ. For template-based repair of $\mathcal{P}$, the concretization needs to ensure that the modified expressions of $\mathcal{P}$ meet the syntactic requirements of the corresponding templates. In the following sections, we describe these two steps in detail.

4.2 Cost-aware Repair of Boolean Programs

Our solution to cost-aware repair of a Boolean program $\mathcal{B}$ relies on automatically computing *inductive assertions*, along with a suitable $\widehat{\mathcal{B}}$, that together

certify the partial correctness of $\widehat{\mathcal{B}}$. In what follows, we explain our adaptation of the method of inductive assertions [52, 90] for cost-aware program repair[4].

Cut-set. Let $N = N_0 \cup \ldots \cup N_t$ be the set of nodes in $\mathcal{G}(\mathcal{B})$, the transition graph representation of $\mathcal{B}$. We define a cut-set $\Lambda \subseteq N$ as a set of nodes, called *cut-points*, such that for every $i \in [0, t]$: (a) $entry_i, exit_i \in \Lambda$, (b) for every edge $(\ell, \varsigma, \ell') \in E_i$ where ς is a procedure `call`, $\ell, \ell' \in \Lambda$, (c) for every edge $(\ell, \varsigma, \ell') \in E_i$ where ς is an `assert` statement, $\ell, \ell' \in \Lambda$, and (d) every cycle in $\mathcal{G}_i$ contains at least one node in Λ. A pair of cut-points ℓ, ℓ' in some $\mathcal{G}_i$ is said to be *adjacent* if every path from ℓ to ℓ' in $\mathcal{G}_i$ contains no other cut-point. A *verification path* is any path from a cut-point to an adjacent cut-point; note that there can be more than one verification path between two adjacent cut-points.

For example, the set $\{\ell_1, \ell_2, \ell_7, exit\}$ is a valid cut-set for Boolean program $\mathcal{B}$ in Fig. 4.1. The verification paths in $\mathcal{G}(\mathcal{B})$ corresponding to this cut-set are as follows:

1. $\ell_1 \xrightarrow{\texttt{assume}\,(b_2)} \ell_2$
2. $\ell_2 \xrightarrow{\texttt{assume}\,(\texttt{T})} \ell_3 \xrightarrow{b_0,b_1,b_2 := *,*,*} \ell_4 \rightarrow \ell_2$
3. $\ell_2 \xrightarrow{\texttt{assume}\,(\texttt{T})} \ell_0 \rightarrow \ell_7$

[4]To certify total correctness, we can additionally compute *ranking functions* and use the method of well-founded sets along with the method of inductive assertions.

4. $\ell_1 \xrightarrow{\texttt{assume}\,(\neg b_2)} \ell_5 \xrightarrow{\texttt{assume}\,(\neg b_1)} \ell_7$

5. $\ell_1 \xrightarrow{\texttt{assume}\,(\neg b_2)} \ell_5 \xrightarrow{\texttt{assume}\,(b_1)} \ell_6 \xrightarrow{b_1,b_1,b_2 := *,*,*} \ell_7$

6. $\ell_7 \xrightarrow{\texttt{assert}\,(\neg b_0)} exit^5$

Inductive assertions. We denote an inductive assertion associated with cut-point ℓ in Λ by $\mathcal{I}_\ell$. Informally, an inductive assertion $\mathcal{I}_\ell$ has the property that whenever control reaches ℓ in any program execution, $\mathcal{I}_\ell$ must be `true` for the current values of the variables in scope. Thus, for a Boolean program, an inductive assertion $\mathcal{I}_\ell$ is in general a Boolean formula over the variables whose scope includes ℓ. To be precise, $\mathcal{I}_\ell$ is a Boolean formula over $V_s[\ell]$, where $V_s[\ell]$ denotes an ℓ^{th} copy of the subset V_s of the program variables, with $V_s = GV \cup formal(\ell)$ if $\ell \in \{exit_1, \ldots, exit_t\}$, and $V_s = inscope(\ell)$ otherwise. Thus, except for the `main` procedure, the inductive assertions at the exit nodes of all procedures *exclude* the local variables declared in the procedure. Let $\mathcal{I}_\Lambda$ denote the set of inductive assertions associated with all the cut-points in Λ.

Verification conditions. A popular approach to verification of sequential, imperative programs is to compute $\mathcal{I}_\Lambda$ such that $\mathcal{I}_\Lambda$ satisfies a set of constraints called *verification conditions*. Let π be a verification path in $\mathcal{G}_i$, from cut-point ℓ to adjacent cut-point ℓ'. The verification condition corresponding to

[5]Labeling this edge with $\texttt{assert}\,(\neg b_0)$ is a slight abuse of the semantics of an assert statement. Our justification is that the constraints formulated later in this section require that the assertion is `true` whenever control reaches location ℓ_7 in an execution path.

π, denoted $VC(\pi)$, is essentially the Hoare triple $\langle \mathcal{I}_\ell \rangle\, stmt(\pi)\, \langle \mathcal{I}_{\ell'} \rangle$, where $stmt(\pi)$ is the sequence of statements labeling π. When $\mathcal{I}_\ell$, $\mathcal{I}_{\ell'}$ are *unknown*, $VC(\pi)$ can be seen as a constraint encoding all possible solutions for $\mathcal{I}_\ell$, $\mathcal{I}_{\ell'}$ such that: every program execution along path π, starting from a set of variable valuations satisfying $\mathcal{I}_\ell$, terminates in a set of variable valuations satisfying $\mathcal{I}_{\ell'}$. Note that the definitions of cut-sets and adjacent cut-points ensure that we do not have to worry about non-termination along verification paths.

From Lem. 3.3.1 we know that the Hoare triple $\langle \mathcal{I}_\ell \rangle\, stmt(\pi)\, \langle \mathcal{I}_{\ell'} \rangle$ can be defined using weakest preconditions or strongest postconditions. In this chapter, as we will see shortly, we find it convenient to use strongest postconditions.

Program verification using the inductive assertions method. Given a program $\mathcal{B}$ annotated with assertions, and a set Λ of cut-points, $\mathcal{B}$ is partially correct if one can compute a set $\mathcal{I}_\Lambda$ of inductive assertions such that: for every verification path π between every pair ℓ, ℓ' of adjacent cut-points in $\mathcal{G}(\mathcal{B})$, $VC(\pi)$ is valid.

Cost-aware repairability conditions. Let $\mathcal{C} : \bigcup_{i=0}^{t} N_i \to \mathbb{N}$ be a function mapping locations to costs. We find it convenient to use $\mathcal{C}_\ell$ to denote the value $\mathcal{C}(\ell)$ at location ℓ. We set $\mathcal{I}_{entry_0} = \texttt{true}$ and $\mathcal{C}_\ell = 0$ if $\ell \in \{entry_0, \ldots, entry_t\}$. Informally, $\mathcal{C}_\ell$ with $\ell \in N_i$ can be seen as recording the cumulative cost of applying a sequence of update schemas to the statements in procedure F_i from location $entry_i$ to ℓ. Thus, for a specific update function $\mathbb{R}$ with cost function

c, $\mathcal{C}_{exit_0}$ records the total cost $Cost_c(\mathbb{R})$ of performing an $\mathbb{R}$-update of the program. Given a verification path π in $\mathcal{G}_i$, from cut-point ℓ to adjacent cut-point ℓ', we extend the definition of $VC(\pi)$ to define the cost-aware repairability condition corresponding to π, denoted $CRC_{PC}(\pi)$. $CRC_{PC}(\pi)$ can be seen as a constraint encoding all possible solutions for inductive assertions $\mathcal{I}_\ell$, $\mathcal{I}_{\ell'}$ and update functions $\mathbb{R}_{\mathcal{U},\mathcal{L}}$, along with associated functions $\mathcal{C}$, such that: every program execution that proceeds along path π via statements modified by applying the update schemas in $\mathbb{R}_{\mathcal{U},\mathcal{L}}$, starting from a set of variable valuations satisfying $\mathcal{I}_\ell$, terminates in a set of variable valuations satisfying $\mathcal{I}_{\ell'}$, for all nondeterministic choices that the program might make along π.

Before we proceed, recall that $\mathcal{I}_\ell$ is a Boolean formula over $V_s[\ell]$, with $V_s = GV \cup formal(\ell)$ if $\ell \in \{exit_1, \ldots, exit_t\}$, and $V_s = inscope(\ell)$ otherwise. Thus, for all locations $\lambda \neq \ell'$ in verification path π from ℓ to ℓ', $V_s = inscope(\lambda)$. In what follows, the notation $[\![u]\!](stmt(\lambda))$ represents the class of statements that may be obtained by applying update schema u on $stmt(\lambda)$, and is defined for our permissible update schemas in Fig. 4.2. Here, f, f_1, f_2 etc. denote unknown Boolean expressions[6], over the variables in $inscope(\lambda)$. Note that the update schema `assign` $\mapsto$ `assign`, modifies *any* assignment statement, to one that assigns unknown Boolean expressions to *all* variables in V_s.

[6]To keep our exposition simple, we assume that these unknown Boolean expressions are deterministic. However, in our prototype tool (see Sec. 4.4), we also have the ability to compute modified statements with nondeterministic expressions such as $*$ or `choose`(f_1, f_2).

u	$[\![u]\!](stmt(\lambda))$
id	$stmt(\lambda)$
`assign` $\mapsto$ `skip`	`skip`
`assume` $\mapsto$ `skip`	`skip`
`call` $\mapsto$ `skip`	`skip`
`assign` $\mapsto$ `assign`	$b_1, \ldots, b_{\lvert V_s \rvert} := f_1, \ldots, f_{\lvert V_s \rvert}$
`assume` $\mapsto$ `assume`	`assume` f
`call` $\mapsto$ `call`	`call` $F_j(f_1, \ldots, f_k)$, where $stmt(\lambda)$: `call` $F_j(e_1, \ldots, e_k)$

Figure 4.2: Definition of $[\![u]\!](stmt(\lambda))$

We now define $CRC_{PC}(\pi)$. There are three cases to consider.

1. $stmt(\pi)$ does not contain a procedure `call` or `assert` statement:

   Let $\mathcal{A}_\lambda$ denote an assertion associated with location λ in π. $CRC_{PC}(\pi)$ is given by the (conjunction of the) following set of constraints:

$$\begin{aligned} \mathcal{A}_\ell \ &= \mathcal{I}_\ell \\ \mathcal{A}_{\ell'} \ &\Rightarrow \mathcal{I}_{\ell'} \\ \bigwedge_{\ell \preceq \lambda \prec \ell'} \bigwedge_{u \in \mathcal{U}_{stmt(\lambda)}} \mathbb{R}(\lambda) = u \ &\Rightarrow \ \mathcal{C}_{\lambda'} = \mathcal{C}_\lambda + c(u, \lambda) \ \wedge \\ & \qquad \mathcal{A}_{\lambda'} = sp([\![u]\!](stmt(\lambda)), \mathcal{A}_\lambda). \end{aligned} \tag{4.1}$$

   In the above, $\prec$ denotes the natural ordering over the sequence of locations in π with λ, λ' denoting consecutive locations, i.e., $\lambda' \in succ(\lambda)$. The notation $\mathcal{U}_{stmt(\lambda)} \subseteq \mathcal{U}$ denotes the set of all update schemas in $\mathcal{U}$ which may be applied to $stmt(\lambda)$. The notation $sp([\![u]\!](stmt(\lambda)), \mathcal{A}_\lambda)$ denotes the strongest postcondition of the assertion $\mathcal{A}_\lambda$ over the class

of statements $[\![u]\!](stmt(\lambda))$. While the strongest postcondition computation described below is similar to the presentation in Sec. 3.2.2, there are some differences. Here, we use multiple variable copies - a copy $V_s[\lambda]$ for each location λ in π - instead of just the two variable copies $V_s[curr]$, $V_s[next]$ used in Sec. 3.2.2. Note that we could have expressed the strongest postcondition computation here using just two variable copies as well. The choice to have multiple variable copies reflects the design choice that was made in our prototype tool.

Let us assume that $\mathcal{A}_\lambda$ is a Boolean formula of the form[7]:

$$\mathcal{A}_\lambda = \rho[\ell, \grave{\lambda}] \wedge \bigwedge_{b \in V_s} b[\lambda] = \xi[\grave{\lambda}], \tag{4.2}$$

where $\grave{\lambda}$, λ are consecutive locations in π with $\lambda \in succ(\grave{\lambda})$, $\rho[\ell, \grave{\lambda}]$ is a Boolean expression over all variable copies $V_s[\mu]$, $\ell \preceq \mu \preceq \grave{\lambda}$, representing the path condition imposed by the program control-flow, and $\xi[\ell]$ is a Boolean expression over $V_s[\grave{\lambda}]$ representing the λ^{th} copy of each variable b in terms of the $\grave{\lambda}^{th}$ copy of the program variables. Note that $\mathcal{A}_\ell = \mathcal{I}_\ell$ is of the form $\rho[\ell]$.

Given $\mathcal{A}_\lambda$ of the form in (4.2), $sp([\![u]\!](stmt(\lambda)), \mathcal{A}_\lambda)$ is defined in Fig. 4.3. Observe that $sp([\![u]\!](stmt(\lambda)), \mathcal{A}_\lambda)$ is a Boolean formula of the same form as (4.2), over variable copies from $V_s[\ell]$ to $V_s[\lambda']$. For the entries `assume` g

[7]In general, $\mathcal{A}_\lambda$ is a disjunction over Boolean formulas of this form; $sp([\![u]\!](stmt(\lambda)), \mathcal{A}_\lambda)$ can then be obtained by computing a disjunction over the strongest postconditions obtained by propagating each such Boolean formula through $[\![u]\!](stmt(\lambda)$ using the rules in Fig. 4.3.

$[\![u]\!](stmt(\lambda))$	$sp([\![u]\!](stmt(\lambda)), \mathcal{A}_\lambda)$
`skip` `goto`	$\rho[\ell, \grave{\lambda}] \wedge \bigwedge_{b \in V_s} b[\lambda'] = b[\lambda]$
`assume` g	$g[\lambda] \wedge \rho[\ell, \grave{\lambda}] \wedge \bigwedge_{b \in V_s} b[\lambda'] = b[\lambda]$
`assume` f	$f[\lambda] \wedge \rho[\ell, \grave{\lambda}] \wedge \bigwedge_{b \in V_s} b[\lambda'] = b[\lambda]$
$b_1, \ldots, b_m := e_1, \ldots, e_m$	$\rho[\ell, \grave{\lambda}] \wedge \bigwedge_{b_i \in V_s, i \in [1,m]} b_i[\lambda'] = e_i[\lambda] \wedge$ $\bigwedge_{b_i \in V_s, i \notin [1,m]} b_i[\lambda'] = b_i[\lambda]$
$b_1, \ldots, b_{\lvert V_s \rvert} := f_1, \ldots, f_{\lvert V_s \rvert}$	$\rho[\ell, \grave{\lambda}] \wedge \bigwedge_{b_i \in V_s} b_i[\lambda'] = f_i[\lambda]$

Figure 4.3: Definition of $sp([\![u]\!](stmt(\lambda)), \mathcal{A}_\lambda)$

and $b_1, \ldots, b_m := e_1, \ldots, e_m$, the expressions $g, e_1, \ldots, e_m$ are *known* beforehand (these entries correspond to $u = id$). For the entries `assume` f and $b_1, \ldots, b_{|V_s|} := f_1, \ldots, f_{|V_s|}$, the expressions $f, f_1, \ldots, f_{|V_s|}$ are *unknown* (these entries correspond to $u =$ `assume` $\mapsto$ `assume` and $u =$ `assign` $\mapsto$ `assign`, respectively). Notation such as $f[\lambda]$ denotes that f is an unknown Boolean expression over $V_s[\lambda]$. For nondeterministic expressions in the RHS of an assignment statement $b_1, \ldots, b_m := e_1, \ldots, e_m$, the strongest postcondition is computed as the disjunction of the strongest postconditions over all possible assignment statements obtained by substituting each $*$ expression with either `false` or `true`.

Thus, to summarize, the set of constraints in (4.1) encodes all $\mathcal{I}_\ell$, $\mathcal{C}_\ell$, $\mathcal{I}_{\ell'}$,

$\mathcal{C}_{\ell'}$ and $\mathbb{R}_{\mathcal{U},\mathcal{L}}$ such that: if $\mathbb{R}_{\mathcal{U},\mathcal{L}}$ is applied to the sequence of statements $stmt(\pi)$ to get some modified sequence of statements, say $\widehat{stmt}(\pi)$, and program execution proceeds along $\widehat{stmt}(\pi)$, then $sp(\widehat{stmt}(\pi), \mathcal{I}_\ell) \Rightarrow \mathcal{I}_{\ell'}$, and $\mathcal{C}_{\ell'}$ equals the cumulative modification cost, counting up from $\mathcal{C}_\ell$.

2. $stmt(\pi)$ contains a procedure `call`, say `call` $F_j(e_1, \ldots, e_k)$:

   The path π, given by $(\ell, \texttt{call}\ F_j(e_1, \ldots, e_k), \ell')$, is a verification path of length 1. Suppose the formal parameters of F_j are $b_1, \ldots, b_k$. $CRC_{PC}(\pi)$ is then given by the following set of constraints:

$$
\begin{aligned}
\mathbb{R} = id \ \ &\Rightarrow \mathcal{C}_{\ell'} = \mathcal{C}_\ell + \mathcal{C}_{exit_j} \ \wedge \\
&\quad \mathcal{I}_\ell \ \Rightarrow \ \mathcal{I}_{entry_j}[\bigwedge_{i\in[1,k]} b_i[entry_j]/e_i[\ell]] \ \wedge \\
&\quad \mathcal{I}_{exit_j}[\bigwedge_{i\in[1,k]} b_i[exit_j]/e_i[\ell']] \ \Rightarrow \ \mathcal{I}_{\ell'} \\
\mathbb{R} = \texttt{call} \mapsto \texttt{skip} \ \ &\Rightarrow \mathcal{C}_{\ell'} = \mathcal{C}_\ell + c(\texttt{call} \mapsto \texttt{skip}, \ell) \ \wedge \\
&\quad \mathcal{I}_{\ell'} = \mathcal{I}_\ell[\bigwedge_{i\in[1,k]} b_i[\ell]/b_i[\ell']] \\
\mathbb{R} = \texttt{call} \mapsto \texttt{call} \ \ &\Rightarrow \mathcal{C}_{\ell'} = \mathcal{C}_\ell + \mathcal{C}_{exit_j} + c(\texttt{call} \mapsto \texttt{call}, \ell) \ \wedge \\
&\quad \mathcal{I}_\ell \ \Rightarrow \ \mathcal{I}_{entry_j}[\bigwedge_{i\in[1,k]} b_i[entry_j]/f_i[\ell]] \wedge \\
&\quad \mathcal{I}_{exit_j}[\bigwedge_{i\in[1,k]} b_i[exit_j]/f_i[\ell']] \ \Rightarrow \ \mathcal{I}_{\ell'}
\end{aligned}
\tag{4.3}
$$

   For $\mathbb{R} = id$, the constraints involve replacing the $entry_j^{th}$, $exit_j^{th}$ copies of the formal parameters in $\mathcal{I}_{entry_j}$, $\mathcal{I}_{exit_j}$ with the corresponding actual parameters $e_1, \ldots, e_k$ expressed over the ℓ^{th}, ℓ'^{th} copies of the program

variables, respectively. For $\mathbb{R} = \texttt{call} \mapsto \texttt{call}$, a similar substitution is performed, except the actual parameters are unknown expressions $f_1, \ldots, f_k$. Finally, for $\mathbb{R} = \texttt{call} \mapsto \texttt{skip}$, the inductive assertion essentially stays the same, with variable copies appropriately adjusted. $\mathcal{C}_{\ell'}$ is in general the sum of $\mathcal{C}_\ell$, the cumulative modification cost $\mathcal{C}_{exit_j}$ of procedure F_j, and the cost of applying the update schema in question.

3. $stmt(\pi)$ contains an `assert` statement, say $\texttt{assert}\, g$:

   Again, π, given by $(\ell, \texttt{assert}\, g, \ell')$, is a verification path of length 1, and $CRC_{PC}(\pi)$ is given by the following set of constraints:

$$\begin{aligned} \mathcal{I}_\ell[\bigwedge_{i\in[1,|V_s|]} b_i[\ell]/b_i[tmp]] &\Rightarrow g[\bigwedge_{i\in[1,|V_s|]} b_i/b_i[tmp]] \\ \mathcal{I}_\ell[\bigwedge_{i\in[1,|V_s|]} b_i[\ell]/b_i[tmp]] &\Rightarrow \mathcal{I}_{\ell'}[\bigwedge_{i\in[1,|V_s|]} b_i[\ell']/b_i[tmp]] \\ \mathcal{C}_{\ell'} &= \mathcal{C}_\ell. \end{aligned}$$

   In the above, we uniformly convert the expressions $\mathcal{I}_\ell$, g and $\mathcal{I}_{\ell'}$ into expressions over some temporary copy of the program variables to enable checking the implications (informally, these implications are $\mathcal{I}_\ell \Rightarrow g$ and $\mathcal{I}_\ell \Rightarrow \mathcal{I}_{\ell'}$).

Cost-aware Boolean program repair. Given a cut-set Λ of $\mathcal{G}(\mathcal{B})$, let Π_Λ be the set of all verification paths between every pair of adjacent cut-points in Λ. Given incorrect program $\mathcal{B}$ annotated with assertions, the set $\mathcal{U}$, cost function c and repair budget δ, we say $\mathcal{B}$ is *repairable within budget* δ if given

a cut-set Λ in $\mathcal{G}$, one can compute a set $\mathcal{I}_\Lambda$ of inductive assertions, an update function $\mathbb{R}$, along with models for all unknown expressions associated with applications of update schemas in $\mathbb{R}$, and the valuations of a cumulative-cost-recording function $\mathcal{C}$ such that: $\mathcal{C}_{exit_0} \leq \delta$, for every verification path $\pi \in \Pi_\Lambda$, $CRC_{PC}(\pi)$ is valid and some other constraints are met. Mathematically, $\mathcal{B}$ is repairable within budget δ if the following formula is `true`:

$$\exists Unknown\, \forall Var: \;\; \mathcal{C}_{exit_0} \leq \delta \;\wedge\; \bigwedge_{\pi \in \Pi_\Lambda} CRC_{PC}(\pi) \;\wedge\; AssumeConstraints, \tag{4.4}$$

where $Unknown$ is the set of all unknowns and Var is the set of all Boolean program variables and their copies used in encoding each $CRC_{PC}(\pi)$. The set of unknowns includes the inductive assertions in $\mathcal{I}_\Lambda$, update function $\mathbb{R}$, unknown expressions f, f_1 etc. associated with applying the update schemas in $\mathbb{R}$ and valuations at each program location of the function $\mathcal{C}$. Finally, $AssumeConstraints$ ensures that any modifications to the guards of `assume` statements corresponding to the same conditional statement are consistent. Thus, for every pair of *updated* `assume` (f_1), `assume` (f_2) statements labeling edges starting from the same node in the transition graph, the uninterpreted functions f_1, f_2 are constrained to satisfy $f_1 = \neg f_2$.

If the above formula is `true`, then we can extract models for all the unknowns from the witness to the satisfiability of the formula: $\forall Var : \mathcal{C}_{exit_0} \leq \delta \wedge \bigwedge_{\pi \in \Pi_\Lambda} CRC_{PC}(\pi) \wedge AssumeConstraints$. In particular, we can extract an $\mathbb{R}$ and the corresponding modified statements to yield a correct Boolean

program $\widehat{\mathcal{B}}$. The following theorem states the correctness and completeness of the above algorithm for repairing Boolean programs for partial correctness.

Theorem 4.2.1. *Given an incorrect Boolean program $\mathcal{B}$ annotated with assertions, the set of $\mathcal{U}$ of permissible, statement-level update schemas given by:*

$$\mathcal{U} = \{id, \texttt{assign} \mapsto \texttt{assign}, \texttt{assign} \mapsto \texttt{skip}, \texttt{assume} \mapsto \texttt{assume},$$
$$\texttt{call} \mapsto \texttt{call}, \texttt{call} \mapsto \texttt{skip}\},$$

a cost function c and repair budget δ,

1. *if there exists a partially correct Boolean program $\widehat{\mathcal{B}}$ such that $\widehat{\mathcal{B}}$ is an $\mathbb{R}$-update of $\mathcal{B}$ for some $\mathbb{R}$ with $Cost_c(\mathbb{R}) \leq \delta$, the above extension of the method of inductive assertions finds one such $\widehat{\mathcal{B}}$; if there exists a unique $\widehat{\mathcal{B}}$, then the method finds it,*

2. *if the above method finds a $\widehat{\mathcal{B}}$, then it can be guaranteed that $\widehat{\mathcal{B}}$ is partially correct and that there exists $\mathbb{R}$ such that $\widehat{\mathcal{B}}$ is an $\mathbb{R}$-update of $\mathcal{B}$ and $Cost_c(\mathbb{R}) \leq \delta$.*

Proof. Note that the formula (4.4) is a $\exists\forall$ formula over Boolean variables (Boolean program variables and their copies), unknown Boolean expressions over these Boolean variables (inductive assertions and expressions in modified program statements), sequences of update schemas (update functions) and corresponding sequences of integer costs (valuations of $\mathcal{C}$). The number of Boolean variables is finite and hence, the number of unknown Boolean expressions over them is finite. There are a finite number of update functions drawn

from finite sequences of update schemas in the finite set $\mathcal{U}$, and a corresponding finite number of $\mathcal{C}$ functions, with $\mathcal{C}_{entry_0}$ set to 0. Besides these (4.4) includes Boolean operators, the + operator and a finite number of integer constants (corresponding to the cost function c). Clearly, the truth of the formula in (4.4) is decidable. In particular, the formula has a finite number of models.

Given a set $\mathcal{U}$ of update schemas, the completeness of our method follows from the completeness of Floyd's inductive assertions method and the decidability of the formula in (4.4).

The soundness of our method follows from the soundness of Floyd's inductive assertions method. □

4.3 Concretization

We now present the second step in our framework for computing a concrete repaired program $\widehat{\mathcal{P}}$. In what follows, we assume that we have already extracted models for $\widehat{\mathcal{B}}$ and $\mathcal{I}_\Lambda$. Recall that γ denotes the mapping of Boolean variables to their respective predicates: for each $i \in [1, |V(\mathcal{B})|]$, $\gamma(b_i) = \phi_i$. The mapping γ can be extended in a standard way to map expressions over the Boolean variables in $V(\mathcal{B})$ to expressions over the concrete program variables in $V(\mathcal{P})$.

Concretization of $\widehat{\mathcal{B}}$. The goal of concretization of a repaired Boolean program $\widehat{\mathcal{B}}$ is to compute a corresponding repaired concrete program $\widehat{\mathcal{P}}$. This

involves computing a mapping, denoted Γ, from each modified statement of $\widehat{\mathcal{B}}$ into a corresponding modified statement in the concrete program. In what follows, we define Γ for each type of modified statement in $\widehat{\mathcal{B}}$. Let us fix our attention on a statement at location ℓ, with $V_s(\mathcal{B})$, $V_s(\mathcal{P})$ denoting the set of abstract, concrete program variables, respectively, whose scope includes ℓ. Let $r = |V_s(\mathcal{B})|$ and $q = |V_s(\mathcal{P})|$.

1. $\Gamma(\texttt{skip}) = \texttt{skip}$

2. $\Gamma(\texttt{assume}\,(g)) = \texttt{assume}\,(\gamma(g))$

3. $\Gamma(\texttt{call}\ F_j(e_1, \ldots, e_k) = \texttt{call}\ F_j(\gamma(e_1), \ldots, \gamma(e_k))$

4. The definition of Γ for an assignment statement is non-trivial. In fact, in this case, Γ may be the empty set, or may contain multiple concrete assignment statements.

   We say that an assignment statement $b_1, \ldots, b_r := e_1, \ldots, e_r$ in $\mathcal{B}$ is *concretizable* if one can compute expressions $f_1, \ldots, f_q$ over $V_s(\mathcal{P})$, of the same type as the concrete program variables $v_1, \ldots, v_q$ in $V_s(\mathcal{P})$, respectively, such that a certain constraint is valid. To be precise, $b_1, \ldots, b_r := e_1, \ldots, e_r$ in $\mathcal{B}$ is concretizable if the following formula is `true`:

   $$\exists f_1, \ldots, f_q\ \forall v_1, \ldots, v_q : \bigwedge_{i=1}^{r} \gamma(b_i)[v_1/f_1, \ldots, v_q/f_q] \ = \ \gamma(e_i) \qquad (4.5)$$

   Each quantifier-free constraint $\gamma(b_i)[v_1/f_1, \ldots, v_q/f_q] \ = \ \gamma(e_i)$ above essentially expresses the concretization of the abstract assignment $b_i = e_i$.

The substitutions $v_1/f_1, \ldots, v_q/f_q$ reflect the *new* values of the concrete program variables after the concrete assignment $v_1, \ldots, v_q := f_1, \ldots, f_q$.

If the above formula is `true`, we can extract models $expr_1, \ldots, expr_q$ for $f_1, \ldots, f_q$, respectively, from the witness to the satisfiability of the inner $\forall$-formula. We then say:

$$v_1, \ldots, v_q := expr_1, \ldots, expr_q \ \in \ \Gamma(b_1, \ldots, b_r := e_1, \ldots, e_r).$$

Note that, in practice, for some $i \in [1, q]$, $expr_i$ may be equivalent to v_i, thereby generating a redundant assignment $v_i := v_i$. The parallel assignment can then be compressed by eliminating each redundant assignment. In fact, it may be possible to infer some such v_i without using (4.5) by analyzing the dependencies of concrete program variables on the predicates in $\{\phi_1, \ldots, \phi_r\}$ that are actually affected by the Boolean assignment in question; this exercise is beyond the current scope of this work.

Template-based concretization of $\widehat{\mathcal{B}}$. Recall that $\mathbb{E}_{\mathcal{T},\mathcal{L}}(\ell)$, associated with location ℓ, denotes a user-supplied template from $\mathcal{T}$, specifying the desired syntax of the expressions in any concrete modified statement at ℓ. Henceforth, we use the shorthand $\mathbb{E}(\ell)$ for $\mathbb{E}_{\mathcal{T},\mathcal{L}}(\ell)$. We find it helpful to illustrate template-based concretization using an example template. Let us assume that for each concrete program variable $v \in V(\mathcal{P})$, $v \in \mathbb{N} \cup \mathbb{R}$. We fix $\mathbb{E}(\ell)$ to (Boolean-valued) linear arithmetic expressions over the program variables, of

the form $c_0 + \Sigma_{p=1}^{q} c_p * v_p \leq 0$, for `assume` and `call` statements, and (integer or real-valued) linear arithmetic *terms* over the program variables, of the form $c_0 + \Sigma_{p=1}^{q} c_p * v_p$, for assignment statements. Let us assume that the parameters $c_0, c_1, \ldots, c_q \in \mathbb{R}$. Given $\mathbb{E}(\ell)$, let $\Gamma_{\mathbb{E}(\ell)}$ denote the mapping of abstract statements into concrete statements compatible with $\mathbb{E}(\ell)$. We can define $\Gamma_{\mathbb{E}(\ell)}$ for each type of modified statement in $\widehat{\mathcal{B}}$ as shown below. The basic idea is to compute suitable values for the template parameters $c_0, \ldots, c_q$ that satisfy certain constraints. Note that, in general, $\Gamma_{\mathbb{E}(\ell)}$ may be the empty set, or may contain multiple concrete statements.

1. $\Gamma_{\mathbb{E}(\ell)}(\texttt{skip}) = \texttt{skip}$

2. The statement $\texttt{assume}\,(g)$ is concretizable if the following formula is `true`:

$$\exists c, \ldots, c_q \; \forall v_1, \ldots, v_q : \;\; (c_0 + \Sigma_{p=1}^{q} c_p * v_p \leq 0) \;=\; \gamma(g). \tag{4.6}$$

   If the above formula is `true`, we extract values from the witness to the satisfiability of the inner $\forall$-formula, and say,

$$c_0 + \Sigma_{p=1}^{q} c_p * v_p \leq 0 \;\in\; \Gamma_{\mathbb{E}(\ell)}(\texttt{assume}\,(g)).$$

3. Similarly, the statement $\texttt{call}\; F_j(e_1, \ldots, e_k)$ is concretizable if the following formula is `true`:

$$\exists c_{1,0}, \ldots, c_{k,q} \; \forall v_1, \ldots, v_q : \;\; \bigwedge_{i=1}^{k} ((c_{i,0} + \Sigma_{p=1}^{q} c_{i,p} * v_p \leq 0) \;=\; \gamma(e_i)).$$

If the above formula is `true`, we can extract values from the witness to the satisfiability of the inner ∀-formula to generate a concrete `call` statement in $\Gamma_{\mathbb{E}(\ell)}(\texttt{call}\ F_j(e_1, \ldots, e_k))$.

4. The statement $b_1, \ldots, b_r := e_1, \ldots, e_r$ is concretizable if the formula in (4.7) is `true`. For convenience, let $h_j = c_{j,0} + \Sigma_{p=1}^{q} c_{j,p} * v_p$, for $j \in [1, q]$.

$$\exists c_{1,0}, \ldots, c_{r,q}\ \forall v_1, \ldots, v_q : \bigwedge_{i=1}^{r} \gamma(b_i)[v_1/h_1, \ldots, v_q/h_q] \ = \ \gamma(e_i). \quad (4.7)$$

If the above formula is `true`, we can extract values from the witness to the satisfiability of the inner ∀-formula to generate a concrete assignment statement in $\Gamma_{\mathbb{E}(\ell)}(b_1, \ldots, b_r := e_1, \ldots, e_r)$.

Concretization of inductive assertions. The concretization of each inductive assertion $\mathcal{I}_\ell \in \mathcal{I}_\Lambda$ is simply $\gamma(\mathcal{I}_\ell)$.

4.4 Experiments with a Prototype Tool

We have built a prototype tool for repairing Boolean programs. The tool accepts Boolean programs generated by the predicate abstraction tool SATABS (version 3.2) [26] from sequential C programs. In our experience, we found that for C programs with multiple procedures, SATABS generates (single procedure) Boolean programs with all procedure calls inlined within the calling procedure. Hence, we only perform intraprocedural analysis in this version of our tool. The set of update schemas handled currently is $\{id, \texttt{assign} \rightarrow \texttt{assign}, \texttt{assume} \rightarrow \texttt{assume}\}$; we do not permit statement deletions. We set the costs $c(\texttt{assign} \rightarrow \texttt{assign}, \ell)$ and $c(\texttt{assume} \rightarrow \texttt{assume}, \ell)$ to

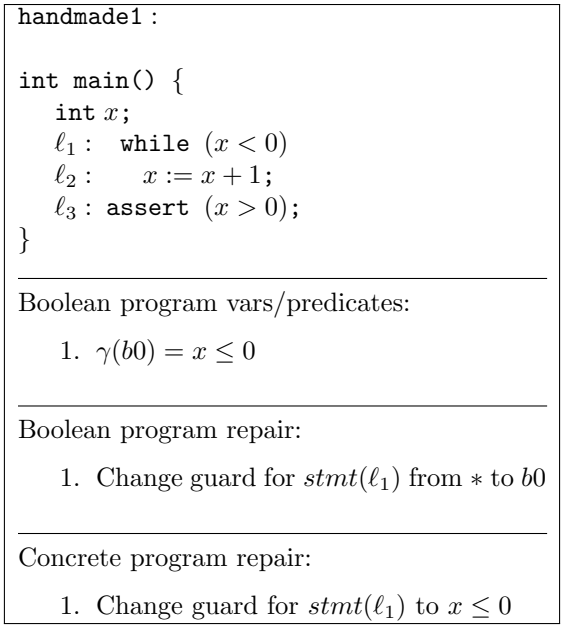


Figure 4.4: Repairing program `handmade1`

some large number for every location ℓ where we wish to disallow statement modifications, and to 1 for all other locations. We initialize the tool with a repair budget of 1. We also provide the tool with a cut-set of locations for its Boolean program input.

Given the above, the tool automatically generates an SMT query corresponding to the inner $\forall$-formula in (4.4). When generating this repairability query, for update schemas involving expression modifications, we stipulate every deterministic Boolean expression g be modified into an *unknown* deterministic Boolean expression f (as described in Fig. 4.2), and every nondeterministic Boolean expression be modified into an unknown nondeterministic expression of the form $\texttt{choose}(f_1, f_2)$. The SMT query is then fed to the

`handmade2` :

```
int main() {
   int x;
   ℓ1 : if (x ≤ 0)
   ℓ2 :    while (x < 0){
   ℓ3 :      x := x + 2;
   ℓ4 :      skip;
           }
        else
   ℓ5 :    if (x == 1)
   ℓ6 :      x := x − 1;
   ℓ7 : assert (x > 1);
}
```

Boolean program vars/predicates:

1. $\gamma(b0) \;=\; x \leq 1$
2. $\gamma(b1) \;=\; x == 1$
3. $\gamma(b2) \;=\; x \leq 0$

Boolean program repair:

1. Change guard for $stmt(\ell_1)$ from $b2$ to $b0 \vee b1 \vee \neg b2$
2. Change guard for $stmt(\ell_2)$ from $*$ to $b0 \vee b1 \vee b2$

Concrete program repair:

1. Change guard for $stmt(\ell_1)$ to `true`
2. Change guard for $stmt(\ell_2)$ to $x \leq 1$

Figure 4.5: Repairing program `handmade2`

SMT-solver Z3 (version 4.3.1) [94]. The solver either declares the formula to be satisfiable, and provides models for all the unknowns, or declares the formula to be unsatisfiable. In the latter case, we can choose to increase the repair budget by 1, and repeat the process.

Once the solver provides models for all the unknowns, we can extract

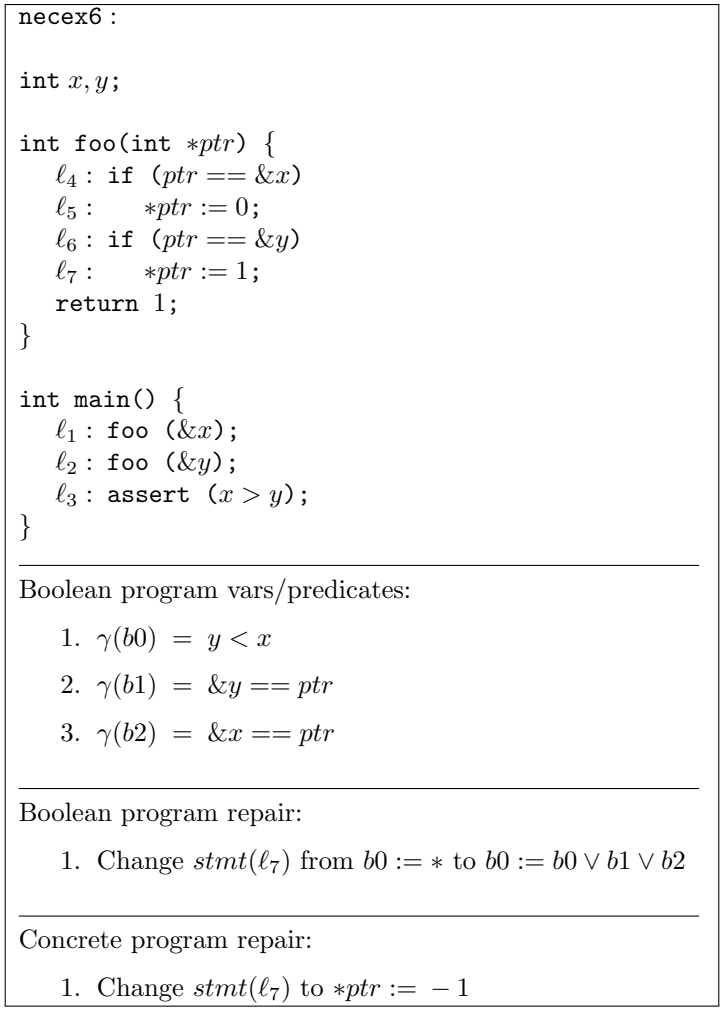


Figure 4.6: Repairing program `necex6`

a repaired Boolean program. Currently, the next step — concretization — is only partly automated. For assignment statements, we manually formulate SMT queries corresponding to the inner ∀-formula in (4.5), and feed these queries to Z3. If the relevant queries are found to be satisfiable, we can obtain a repaired C program. If the queries are unsatisfiable, we attempt template-

based concretization using linear-arithmetic templates. We manually formulate SMT queries corresponding to the inner $\forall$-formulas in (4.6) and (4.7), and call Z3. In some of our experiments, we allowed ourselves a degree of flexibility in guiding the solver to choose the right template parameters.

We describe our experiments with four C programs in Fig. 4.4, Fig. 4.5, Fig. 4.6 and Fig. 4.7. The first two programs are handmade, with the second one being the same as the one shown in Fig. 4.1. The next two programs are mutations of two programs drawn from the NEC Laboratories Static Analysis Benchmarks [97].

We emphasize that the repairs for the respective Boolean programs (not shown here due to lack of space) are obtained automatically. The concretization of the repaired Boolean program in Fig. 4.4 was trivial – it only involved concretizing the guard $b0$ corresponding to the statement at location ℓ_1. Concretization of the repaired Boolean program in Fig. 4.5 involved concretizing two different guards, $b0 \vee b1 \vee \neg b2$ and $b0 \vee b1 \vee b2$, corresponding to the statements at locations ℓ_1 and ℓ_2, respectively. We manually simplified the concretized guards to obtain the concrete guards `true` and $x \leq 1$, respectively. Concretization of the repaired Boolean program in Fig. 4.6 involved concretizing the assignment statement at location ℓ_7. We manually formulated an SMT query corresponding to the formula in (4.5), after simplifying $\gamma(b_0 \vee b_1 \vee b_2)$ to $y < x$ and restricting the LHS of $stmt(\ell_7)$ in the concrete program to remain unchanged. The query was found to be satisfiable, and yielded -1 as the RHS of the assignment statement in the concrete program. We repeated the above

necex14 :

```
int main() {
   int x, y;
   int a[10];
   ℓ1 : x := 1U;
   ℓ2 : while (x ≤ 10U) {
   ℓ3 :     y := 11 − x;
   ℓ4 :     assert (y ≥ 0 ∧ y < 10);
   ℓ5 :     a[y] := −1;
   ℓ6 :     x := x + 1;
        }
}
```

Boolean program vars/predicates:

1. $\gamma(b0) \;=\; y < 0$
2. $\gamma(b1) \;=\; y < 10$

Boolean program repair:

1. Change $stmt(\ell_3)$ from $b0, b1 := *, *$ to $b0, b1 := \texttt{F}, \texttt{T}$

Concrete program repair:

1. Change $stmt(\ell_3)$ to $y := 10 - x$

Figure 4.7: Repairing program `necex14`

exercise to concretize the assignment statement at location ℓ_3 in Fig. 4.7, and obtained $y := 0$ as the repair for the concrete program. Unsatisfied by this repair, we formulated another SMT query corresponding to the formula in (4.7), restricting the RHS of $stmt(\ell_3)$ to the template $-x + c$, where c is unknown. The query was found to be satisfiable, and yielded $c = 10$.

Notice that our tool can repair diverse programs — programs containing loops, multiple procedures and pointer and array variables. Also, our tool can repair programs in multiple locations, and can repair assignment, conditional

Table 4.1: Experimental results

Name	LoC($\mathcal{P}$)	LoC($\mathcal{B}$)	$V(\mathcal{B})$	$\mathcal{B}$-time	Que-time	Sol-time
`handmade1`	6	58	1	0.180s	0.009s	0.012s
`handmade2`	16	53	3	0.304s	0.040s	0.076s
`necex6`	24	66	3	0.288s	0.004s	0.148s
`necex14`	13	60	2	0.212s	0.004s	0.032s
`tcas1`	283	856	49	17m58.675s	0.308s	$>$ 5h

and loop statements.

In Table 4.1, we enumerate the time taken for each individual step involved in generating a repaired Boolean program. The columns labeled LoC($\mathcal{P}$) and LoC($\mathcal{B}$) enumerate the number of lines of code in the original C program and the Boolean program generated by SATABS, respectively. The column labeled $V(\mathcal{B})$ enumerates the number of variables in each Boolean program. The column $\mathcal{B}$-time enumerates the time taken by SATABS to generate each Boolean program, the column Que-time enumerates the time taken by our tool to generate each repairability query and the column Sol-time enumerates the time taken by Z3 to solve the query. The total time taken for the programs `handmade1`, `handmade2`, `necex6` and `necex14` was negligible. The current version of the tool has scalability limitations for Boolean programs with a large number of variables and nondeterministic expressions. This is seen in the last row of the table for an attempted repair of a program from the Traffic Collision Avoidance System (TCAS) benchmark suite [39].

Chapter 5

Bibliographic Notes

There is a diverse body of work on automated repair and synthesis of software programs as well as hardware circuits, geared towards guaranteeing various correctness criteria. In what follows, we mainly review relevant work on automated repair and synthesis of sequential, imperative software programs.

Program repair. Early proposals for program repair were based on computing repairs as winning strategies in finite-state games [75] and pushdown games [57]. The work in [57] was the first to use Boolean programs for repairing infinite-state systems. The approach, however, only targeted partial correctness, and had formidable complexity (doubly exponential in the number of program variables). In contrast, our approach from Chapter 3 (also presented in [109]) targeted repair of (a large subset of) Boolean programs with respect to total correctness, and had tractable complexity (similar to model checking Boolean programs). Both these initial explorations into predicate-abstraction-based repair of infinite-state programs, however, shared two main limitations - a restrictive single-fault model, and failure to address the problem of readability of the generated repairs. These limitations are partially addressed in

a recent abstract interpretation-based approach [87], that can repair multiple, diverse faults in C# programs. While this approach only targets partial correctness, its efficacy within the .NET framework is impressive, and its applicability to other programming environments is worth exploring. The approach presented in Chapter 4 (also presented in [110]) addresses these limitations within an arguably more formal and flexible framework. In particular, our approach presents a sound and complete algorithm for repairing Boolean programs with multiple faults and assertions guided by user-defined update schemas, costs and repair budgets. Our approach has the ability to ensure the readability of the repaired program using user-defined expression templates, and generates a proof of correctness along with a repaired program.

An alternate approach that does not rely on abstract interpretations of concrete programs was proposed recently in [81]. It encodes the repair problem as a constraint-solving problem using symbolic execution, and uses SMT reasoning to search for repairs satisfying user-defined templates. We remark that the templates are needed not only for ensuring readability of the generated repairs, but also for ensuring tractability of their inherently undecidable repair generation query. The technique does not produce an explicit proof of correctness for the repaired program, can only generate repairs for faulty expressions and is limited to handling partial correctness, due to the bounded semantics associated with symbolic execution. While the work presented in Chapter 4 focuses on partial correctness as well, it can be extended to handle total correctness by computing ranking functions along with inductive asser-

tions.

Besides the above, there have been proposals for program repair based on mutations and genetic algorithms [2,30,55], principles drawn from artificial intelligence [20,33,34,135], for UNITY programs [42], programs equipped with contracts [130], Kripke structures [6,135], reactive programs [20,128], digital circuits (see [76] for a survey), and programs with data structure manipulations [35,113,133]. There are also customized program repair engines [112] for grading and feedback generation for programming assignments.

Partial Program Synthesis. Techniques for program completion and synthesis, based on user-supplied program schemas [28], program sketches [115,116] and scaffolds [117], or for restricted domains [82] can also be adapted for use for program repair. Sketching does not handle loops precisely due to its inherent bounded semantics, and does not explicitly generate a proof of correctness. The scaffold or template-based program synthesis framework of [117] is perhaps the closest to our own framework. Given a scaffold consisting of a functional specification, domain constraints for the program expressions, the structure of the program flowgraph and some other resource constraints, the framework attempts to synthesize a program, along with a proof of total correctness consisting of program invariants and ranking functions for loops. The proof objects - invariants and ranking functions - are drawn from a proof domain, chosen by the user. In particular, the framework has the ability to synthesize invariants over predicate abstraction and linear arithmetic domains,

and ranking functions over some restricted numeric domains. We emphasize that our framework is self-reliant and only interacts with a user for the cost function and for improving the readability of the generated repairs; all predicates involved in the generation of the repaired Boolean program and its proof are discovered automatically. In contrast, the framework of [117] heavily relies on users to provide expressive templates and domains for invariants.

Fault Localization. In addition to the above work, a multitude of algorithms [11, 21, 38, 50, 58, 74, 77, 105, 134] have been proposed for error explanation, diagnosis and localization based on analyzing error traces. Often,these techniques try to localize faults by finding correct program traces most similar to an incorrect counterexample trace. Even though many of these techniques operate on the premise of a single-fault assumption, and we believe searching for *repairable* statements or program fragments is a more formal approach to program debugging than many of the proposed fault localization heuristics, some fault localization techniques can definitely be used as a preprocessing step for our framework. Finally, there's a significant body of work on *algorithmic debugging* [111] (see [71] for a survey), which attempts to localize faults in programs by asking a series of questions to a programmer and utilizing the (yes/no) answers to guide the search.

Quantitative Approaches. Quantitative synthesis, whose overall goal, is similar to cost-aware program repair was proposed in [15] in the context of reactive

systems. The repair approaches in [87, 130] have the ability to rank generated repairs based on various criteria. The work in [81] includes a notion of *minimal diagnoses*, which is subsumed by our cost-aware framework.

Part III

Synchronization Synthesis

In Chapter 1, we have seen that error detection and debugging in concurrent programs are particularly challenging tasks. This makes concurrent programs excellent targets for automated program completion, in particular, for synthesis of synchronization code. Following the foundational work in [47], we assert that one can simplify the design and analysis of (shared-memory) concurrent programs by, first, manually writing synchronization-free concurrent programs, followed by, automatically synthesizing the synchronization code necessary for ensuring the programs' correct concurrent behaviour.

In this part of the dissertation, we present a framework for synthesis of synchronization for shared-memory concurrent programs with respect to temporal logic specifications. Our framework generalizes the approach of [47] in several ways. We provide the ability to synthesize more readily-implementable synchronization code based on lower-level primitives such as locks and condition variables. We also extend the approach of [47] to enable synchronization synthesis for more general programs and properties.

In Chapter 6, we present our first generalization. Similar to [47], our framework accepts unsynchronized sequential process *skeletons*, along with a specification in propositional temporal logic of their desired concurrent behaviour, and synthesizes high-level synchronization actions in the form of guarded commands. In addition, our framework has the ability to perform a *correctness-preserving* compilation of guarded commands into coarse-grained or fine-grained synchronization code based on locks and condition variables.

In Chapter 7, we present a framework that supports finite-state concur-

rent programs composed of processes that may have local and shared variables, may have a linear or branching control-flow, may be ongoing or terminating, and may have program-initialized or user-initialized variables. The specification language is an extension of propositional CTL that enables easy specification of safety and liveness properties over control and data variables. The framework also supports synthesis of synchronization at different levels of abstraction and granularity.

We conclude this part with a discussion of related work in Chapter 8.

Chapter 6

Synthesis of Low-level Synchronization

Overview. In this chapter, we present a framework that takes unsynchronized sequential process *skeletons* along with a propositional temporal logic specification of their global concurrent behaviour, and automatically generates a concurrent program with synchronization code ensuring correct global behaviour. The synthesized synchronization code can be coarse-grained or fine-grained, and is based on locks and condition variables. The method is fully automatic, sound and complete.

6.1 Formal Framework

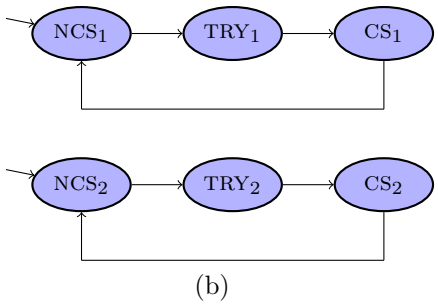
In this section, we define the model of concurrent programs, the synchronization primitives and the specification language(s) relevant to this chapter.

6.1.1 Concurrent Program Model

We focus on nonterminating concurrent programs $\mathcal{P}$, consisting of a fixed, finite number of sequential processes $\mathcal{P}_1, \ldots, \mathcal{P}_K$ running in parallel. Similar to [47], we use process *skeletons* as abstractions of the constituent sequential processes. A process skeleton can be obtained from a process by suppressing

```
P1() {
  while(true) {
    Execute code region NCS1;
    Execute code region TRY1;
    Execute code region CS1;
}}
P2() {
  while(true) {
    Execute code region NCS2;
    Execute code region TRY2;
    Execute code region CS2;
}}
```

(a)



(b)

Figure 6.1: Synchronization-free skeletons of reader $\mathcal{P}_1$, writer $\mathcal{P}_2$

all computation details that are irrelevant for synchronization. This abstraction is justified by the observation that for most concurrent programs, the regions of code in each process responsible for interprocess synchronization can be cleanly isolated from the regions of code in each process responsible for sequential application-oriented computations. This makes it possible to synthesize synchronization without specifying the internal structure and intended application of the regions of sequential code in concurrent programs.

A synchronization-free or unsynchronized process skeleton consists of sequential *code regions*, wherein each code region may in reality represent a (suppressed) sequential block of code. As an example, we refer the reader to the synchronization-free skeletons for a reader process $\mathcal{P}_1$ and a writer process $\mathcal{P}_2$ in the single-reader, single-writer example shown in Fig. 6.1a. Both the reader and writer process skeletons have three sequential code regions — 'noncritical' (NCS), 'trying' (TRY) and 'critical' (CS). The diagrams shown in

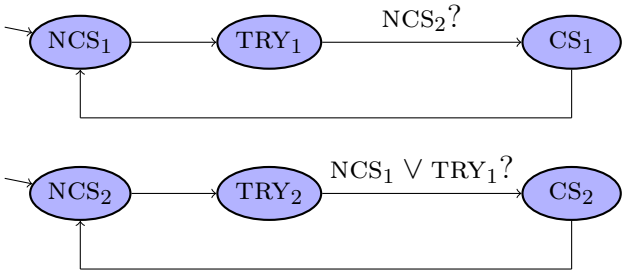


Figure 6.2: Synchronization skeletons $\mathcal{P}_1^s$, $\mathcal{P}_2^s$ of reader $\mathcal{P}_1$, writer $\mathcal{P}_2$.

Fig. 6.1b encode the control flow between these code regions as state-transition diagrams with three states - NCS, TRY and CS. Thus, each node/state in the state-transition diagram representation of a process skeleton corresponds to a unique code region and each edge/transition corresponds to a flow of control between code regions. The initial states NCS_1, NCS_2 of $\mathcal{P}_1$, $\mathcal{P}_2$, respectively are identified with incoming pointers as shown. Note that we focus on non-terminating programs, and hence require that each node in a process has an outgoing edge. We often use states and their corresponding code regions interchangeably. Further note that each code region can always be instantiated with its suppressed sequential code block to obtain the complete process. We only require that each suppressed sequential code block terminates. In this chapter, we use process, skeleton and process skeleton interchangeably.

The *synchronization skeleton* $\mathcal{P}_k^s$ corresponding to an unsynchronized process skeleton $\mathcal{P}_k$ also includes labels on the transitions between the code regions, as shown, for example, in Fig. 6.2. Each transition label is used to enforce some conditional synchronization constraint, and is a *guarded command*

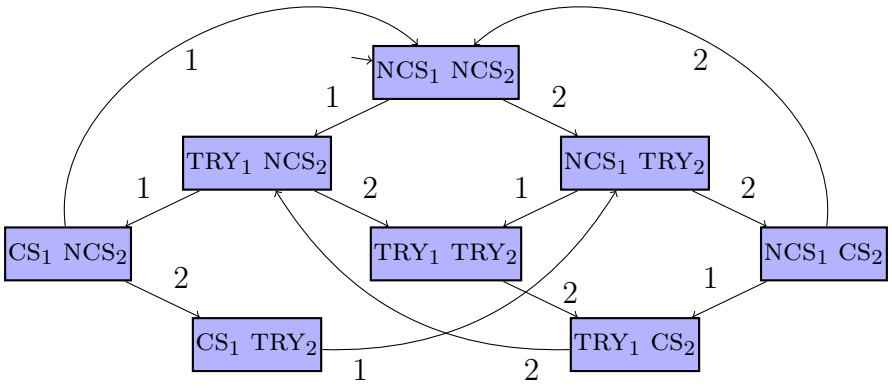


Figure 6.3: The concurrent program corresponding to Fig. 6.2. The edge labels indicate which process's transition is executed next.

of the form $G? \to A$, with an enabling condition G, evaluated atomically, and a corresponding set of actions A to be performed atomically if G evaluates to `true`. An omitted guard is interpreted as `true` in general.

A concurrent program $\mathcal{P} = \mathcal{P}_1 \parallel \ldots \parallel \mathcal{P}_K$ composed of K processes can be viewed as a state-transition diagram[1] with a set of global states and a set of transitions between them (see Fig. 6.3 for a representation of the concurrent program corresponding to the synchronization skeletons in Fig. 6.2). A global state is composed of individual process states (i.e., current code regions) $\mathrm{S}_1, \ldots, \mathrm{S}_K$ and the values of shared synchronization variables $x_1, \ldots, x_m$ (this tuple is often denoted as $\bar{x}$). An initial global state is composed of the initial

[1]To help distinguish between the state-transition diagrams associated with individual processes and concurrent programs, we use oval and rectangular shapes to represent their respective states.

process states and the initial values of the $\bar{x}$ variables. A guard in the synchronization skeleton of a process is a predicate on these global states, and an action is a parallel assignment statement that updates the values of the shared synchronization variables.

We model parallelism in the usual way by the nondeterministic interleaving of the atomic transitions of the synchronization skeletons of the processes. Hence, at each step of the computation, some process with an enabled transition is nondeterministically selected to be executed next. There exists a transition in the concurrent program $\mathcal{P}$ from a current global state $(\mathrm{S}_1, \ldots, \mathrm{S}_k, \ldots, \mathrm{S}_K, x_1, \ldots, x_m)$ to a next state $(\mathrm{S}_1, \ldots, \mathrm{S}'_k, \ldots, \mathrm{S}_K, x'_1, \ldots, x'_m)$ iff there exists a corresponding labeled transition $\mathrm{S}_k \xrightarrow{G?\rightarrow A} \mathrm{S}'_k$ in the synchronization skeleton for process $\mathcal{P}_k$ such that the guard G is `true` in the current global state, and the action A results in the new valuation $x'_1, \ldots, x'_m$ of the shared synchronization variables.

6.1.2 Synchronization Primitives — Locks and Condition Variables

Besides using guarded commands to provide high-level synchronization, we use locks and condition variables as lower-level synchronization primitives in a concurrent program. Each such primitive is meant to be used in a *synchronization region* preceding every code region in a synchronized process skeleton. In this subsection, we define each of these primitives using shared synchronization variables.

To be able to accommodate these primitives, we need to make small

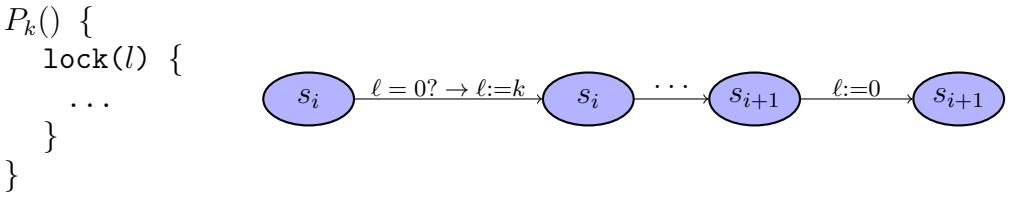


Figure 6.4: Syntax and semantics for `lock`$(\ell)\{\ldots\}$ in process $\mathcal{P}_k$

modifications to our concurrent program model. In particular, we use a Boolean shared variables s_i to label the state corresponding to code region S_i of a process skeleton. The global state of a concurrent program is composed of the values of all these Boolean shared variables, along with the shared synchronization variables as before. The labels on the transitions in a synchronization skeleton are guarded commands as before, but the guards are predicates on the global state comprised of shared variables, and the actions are parallel assignments statements updating the values of all shared variables. There is another important difference in the structure of a synchronization skeleton based on these synchronization primitives. In order to enable expression of multiple lower-level synchronization actions (within a synchronization region) that are equivalent to a single high-level synchronization action, we permit a *sequence* of labeled transitions between two code regions of a process. For this reason, we typically refer to the synchronization skeletons based on locks and condition variables as *refined* synchronization skeletons.

Locks: We express locks syntactically, as shown in Fig. 6.4 (in a manner similar to Java's `synchronized` keyword), wherein '{' denotes *lock acquisition*, '}'

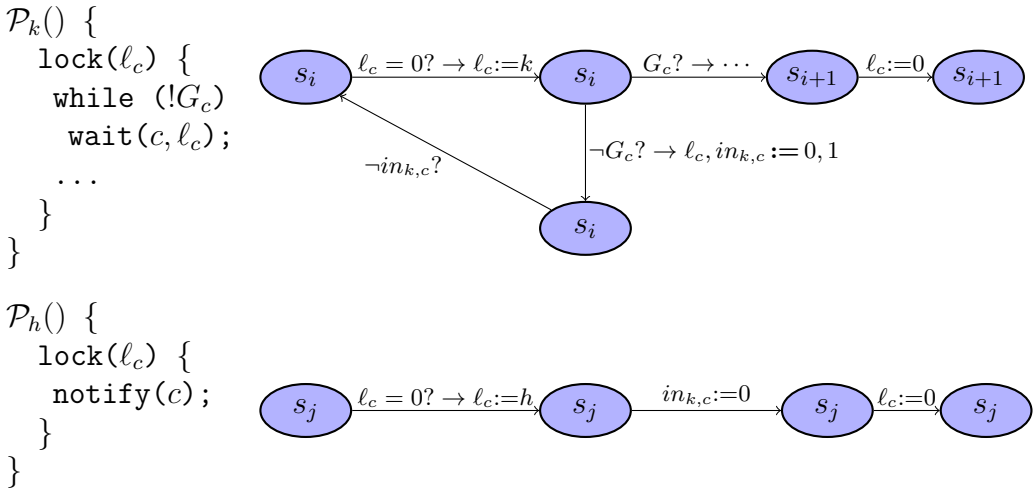


Figure 6.5: Syntax and semantics of `wait(`c,ℓ_c`)` in $\mathcal{P}_k$ and `notify(`c`)` in $\mathcal{P}_h$

denotes *lock release*, and ℓ is a lock variable. The value of the lock variable is assumed to be 0 when the lock is available and k when the lock is acquired by process $\mathcal{P}_k$. The semantics of locks is expressed in the refined synchronization skeleton of process $\mathcal{P}_k$ as shown in Fig. 6.4. Note that, for brevity, we label a state with the (single) binary shared variable which is `true` in it, and leave out all the `false` binary shared variables. Acquisition of lock ℓ in process $\mathcal{P}_k$ is modeled as a labeled transition with the label: $\ell = 0? \to \ell := k$. After acquiring the lock, the process may perform various tasks that possible change its state. Release of lock ℓ is modeled using the transition label: $\ell := 0$.

Condition variables: The syntax and the semantics of the `wait` and `notify` operations on condition variables, assumed in this work, are shown in Fig. 6.5.

Thus, a process $\mathcal{P}_k$ first acquires the lock ℓ_c, associated with condition variable c, and then checks the predicate G_c corresponding to c. If the predicate is `true`, it proceeds with any other task that may need to be performed in the synchronization region, possibly changing the state of $\mathcal{P}_k$, and releases ℓ_c when done. If G_c is `false` $\mathcal{P}_k$ executes `wait(`c,ℓ_c`)`. In this event, $\mathcal{P}_k$ automatically and atomically releases lock ℓ_c and adds itself to a wait queue $q_{k,c}$ of processes blocking/sleeping on condition variable c. Process $\mathcal{P}_k$ can be awakened, or equivalently, removed from $q_{k,c}$ by a `notify(`c`)` operation by some other process $\mathcal{P}_h$. Before executing `notify(`c`)`, process $\mathcal{P}_h$ needs to acquire the lock ℓ_c as well. Upon waking up, $\mathcal{P}_k$ attempts to reacquire lock ℓ_c, and repeats the entire process. We model addition/removal of process $\mathcal{P}_k$ to/from $q_{k,c}$ using a Boolean variable $in_{k,c}$ as shown in Fig. 6.5.

6.1.3 Specification Language(s)

In this chapter, we focus on propositional temporal logics such as CTL^* and its various subsets. The syntax of (propositional) CTL^* formulas, over a set AP of atomic propositions, can be defined inductively using a class of *state formulas*, evaluated in states, and a class of *path formulas*, evaluated over paths as follows:

1. Every atomic proposition $p \in AP$ is a state formula.
2. If ϕ_1, ϕ_2 are state formulas, then so are $\neg\phi_1$ and $\phi_1 \wedge \phi_2$.
3. If ψ is a path formula, then $\mathsf{E}\,\psi$ is a state formula.

4. Any state formula is also a path formula.

5. If ψ_1, ψ_2 are path formulas, then so are $\neg\psi_1$ and $\psi_1 \wedge \psi_2$.

6. If ψ_1, ψ_2 are path formulas, then so are $\mathsf{X}\,\psi_1$, $\mathsf{X}_k\,\psi_1$[2] and $\psi_1\,\mathsf{U}\,\psi_2$.

The set of state formulas generated by the above rules constitute the language CTL^*. We use the following standard abbreviations: $\phi_1 \vee \phi_2$ for $\neg(\neg\phi_1 \wedge \neg\phi_2)$, $\phi_1 \rightarrow \phi_2$ for $\neg\phi_1 \vee \phi_2$, $\phi_1 \leftrightarrow \phi_2$ for $(\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)$, $\mathsf{A}\,\psi$ for $\neg\mathsf{E}\,\neg\psi$, $\mathsf{F}\,\psi$ for $\mathtt{true}\,\mathsf{U}\,\psi$, and $\mathsf{G}\,\psi$ for $\neg\mathsf{F}\,\neg\psi$.

We define the semantics of a CTL^* formula over structures, or *models*, of the form $\mathcal{M} = (S, R, L)$, where S is a set of states, R is a a total, multi-process, binary relation $R = \cup_k R_k$ over S, composed of the transitions R_k of each process $\mathcal{P}_k$, and L is a labeling function that assigns to each state $u \in S$ a set of atomic propositions from AP $\mathtt{true}$ in u. A path in $\mathcal{M}$ is an infinite sequence $\pi = (u_1, u_2, \ldots)$ of states such that $(u_j, u_{j+1}) \in R$, for all $j \geq 1$. We denote by π_j the j^{th} state in π, and by π^j the suffix path $(u_j, u_{j+1}, \ldots)$.

The satisfiability of a CTL^* formula in a state u or path $\pi = (u_1, u_2, \ldots)$ of $\mathcal{M}$ can be defined as follows:

1. $\mathcal{M}, u \models p$ iff $p \in L(u)$, for an atomic proposition p.

2. $\mathcal{M}, u \models \neg\phi$ iff it is not the case that $\mathcal{M}, u \models \phi$.

[2] X_k is a process-indexed nexttime operator, and is typically not included in the definition of CTL^*

3. $\mathcal{M}, u \models \phi_1 \wedge \phi_2$ iff $\mathcal{M}, u \models \phi_1$ and $\mathcal{M}, u \models \phi_2$.

4. $\mathcal{M}, u \models \mathsf{E}\,\psi$ iff for some path π starting at u, $\mathcal{M}, \pi \models \psi$.

5. $\mathcal{M}, \pi \models \phi$ iff $\mathcal{M}, u_1 \models \phi$, for a state formula ϕ.

6. $\mathcal{M}, \pi \models \neg\psi$ iff it is not the case that $\mathcal{M}, \pi \models \psi$.

7. $\mathcal{M}, \pi \models \psi_1 \wedge \psi_2$ iff $\mathcal{M}, \pi \models \psi_1$ and $\mathcal{M}, \pi \models \psi_2$.

8. $\mathcal{M}, \pi \models \mathsf{X}\,\psi$ iff $\mathcal{M}, \pi^2 \models \psi$.

9. $\mathcal{M}, \pi \models \mathsf{X}_k\,\psi$ iff $(u_1, u_2) \in R_k$ and $\mathcal{M}, \pi^2 \models \psi$.

10. $\mathcal{M}, \pi \models \psi_1\, U\, \psi_2$ iff for some $j \geq 1$, $\mathcal{M}, \pi^j \models \psi_2$ and for all $i < j$, $\mathcal{M}, \pi^i \models \psi_1$.

We say $\mathcal{M} \models \phi$, if for every u in a set S_0 of initial states in $\mathcal{M}$, $\mathcal{M}, u \models \phi$.

The language CTL is a subset of CTL^* obtained by restricting the path formulas to just $\mathsf{X}\,\psi_1$, $\mathsf{X}_k\,\psi_1$ and $\psi_1\,\mathsf{U}\,\psi_2$ (i.e., by eliminating syntactic rules (4) and (5)). The language ACTL is the universal fragment of CTL obtained by disallowing the E operator, and the language $\mathsf{ACTL} \setminus \mathsf{X}$ is the universal fragment of CTL without the nexttime operator. Finally, the language LTL is defined by the set of all path formulas generated using the syntactic rules (4), (5) and (6).

Note that a concurrent program $\mathcal{P}$, as defined in Sec. 6.1.1 can be viewed as a model $\mathcal{M} = (S, R, L)$, with the same set of states and transitions

Table 6.1: Specification for single-reader, single-writer problem

Mutual exclusion:	$\mathsf{AG}(\neg(\text{CS}_1 \wedge \text{CS}_2))$.
Absence of starvation for reader, provided writer remains in NCS:	$\mathsf{AG}(\text{TRY}_1 \Rightarrow \mathsf{AF}(\text{CS}_1 \vee \neg\text{NCS}_2))$.
Absence of starvation for writer:	$\mathsf{AG}(\text{TRY}_2 \Rightarrow \mathsf{AF}\,\text{CS}_2)$.
Priority of writer over reader for outstanding requests to enter CS:	$\mathsf{AG}((\text{TRY}_1 \wedge \text{TRY}_2) \Rightarrow \mathsf{A}[\text{TRY}_1\,\mathsf{U}\,\text{CS}_2])$.

as $\mathcal{P}$, and the identity labeling function L that maps a state to itself. Here, the set AP of atomic propositions is composed of all possible individual process states/code regions and values of shared variables. Given a CTL^* specification ϕ, we say $\mathcal{P} \models \phi$ iff for each initial state $u \in S$, $\mathcal{M}, u \models \phi$.

6.2 Motivating Example

Let us revisit the single-reader, single-writer example with the synchronization-free skeletons for the reader process $\mathcal{P}_1$ and the writer process $\mathcal{P}_2$ as shown in Fig. 6.1. The desired set of CTL[3] properties for the concurrent program composed of $\mathcal{P}_1$ and $\mathcal{P}_2$, specifying mutual exclusion and starvation-freedom for the writer, are shown in Table 6.1. Let us denote the conjunction of these properties as ϕ_{spec}. It is easy to see that in the absence of synchronization $\mathcal{P}_1 \parallel \mathcal{P}_2 \not\models \phi_{spec}$. Our goal is to modify $\mathcal{P}_1$ and $\mathcal{P}_2$ by inserting synchronization code such that the resulting concurrent program satisfies ϕ_{spec}.

In step one of our approach, we first use the state-transition diagram representations of $\mathcal{P}_1$ and $\mathcal{P}_2$ to automatically generate a CTL formula $\phi_{\mathcal{P}}$,

[3]Notice that these properties are all in $\mathsf{ACTL} \setminus \mathsf{X}$.

specifying the concurrency model and control-flow of the unsynchronized processes. We then use the methodology of [46, 47] to: synthesize a global model $\mathcal{M}$, based on $\mathcal{P}_1$ and $\mathcal{P}_2$, such that $\mathcal{M} \models \phi_{\mathcal{P}} \wedge \phi_{spec}$ (see Fig. 6.3), and decompose $\mathcal{M}$ to obtain synchronization skeletons $\mathcal{P}_1^s$ and $\mathcal{P}_2^s$ (see Fig. 6.2), with guarded commands labeling the transitions between code regions.

```
main() {
   boolean ncs1 := 1, try1 := 0, cs1 := 0, ncs2 := 1, try2 := 0, cs2 := 0;
   lock ℓ, condition variables cv_cs1, cv_cs2;
   P1^c() ∥ P2^c();
}
```

```
P1^c() {
   while(true) {
     Execute code region NCS1;
     /* Synchronization region */
     lock(ℓ) {
       ncs1, try1 := 0, 1;
       notify(cv_cs2);
     }
     Execute code region TRY1;
     /* Synchronization region */
     lock(ℓ) {
       while (!ncs2)
          wait(cv_cs1, l);
       try1, cs1 := 0, 1;
     }
     Execute code region CS1;
     /* Synchronization region */
     lock(ℓ) {
       cs1, ncs1 := 0, 1;
       notify(cv_cs2);
 }}}
```

```
P2^c() {
   while(true) {
     Execute code region NCS2;
     /* Synchronization region */
     lock(ℓ) {
       ncs2, try2 := 0, 1;
     }
     Execute code region TRY2;
     /* Synchronization region */
     lock(ℓ) {
       while (!(ncs1 ∨ try1))
          wait(cv_cs2, l);
       try2, cs2 := 0, 1;
     }
     Execute code region CS2;
     /* Synchronization region */
     lock(ℓ) {
       cs2, ncs2 := 0, 1;
       notify(cv_cs1);
 }}}
```

Figure 6.6: The concurrent program $\mathcal{P}_1^c \parallel \mathcal{P}_2^c$

In the second step of our approach, we mechanically compile the guarded commands of $\mathcal{P}_1^s$ and $\mathcal{P}_2^s$ into both coarse-grained and fine-grained synchronization code for $\mathcal{P}_1$ and $\mathcal{P}_2$. The resulting concurrent programs are shown in pseudocode in Fig. 6.6 and Fig. 6.7.

As can be seen from the variable declarations of these concurrent pro-

grams, we introduce Boolean shared variables, ncs_1, try_2 etc., to capture the code regions, NCS_1, TRY_2 etc., of each sequential process. We also declare locks and conditions variables for synchronization. The program $\mathcal{P}_1^c \parallel \mathcal{P}_2^c$ in Fig. 6.6 has a coarser level of lock granularity with a single lock ℓ for controlling access to the Boolean shared variables and the two condition variables cv_{cs_1} and cv_{cs_2}. The program $\mathcal{P}_1^f \parallel \mathcal{P}_2^f$ in Fig. 6.7 has a finer level of lock granularity with separate locks, ℓ_{ncs_1}, ℓ_{try_2} etc., for controlling access to the Boolean shared variables, ncs_1, try_2 etc., respectively, and separate locks, $\ell_{cv_{cs1}}$ and $\ell_{cv_{cs2}}$, for the two condition variables cv_{cs_1} and cv_{cs_2}, respectively.

The modifications to each process are restricted to insertion of synchronization regions between the sequential code regions of the process. The implementation of a coarse-grained synchronization region for the reader-writer example involves acquiring the lock ℓ and checking if the guard G for entering the next code region is enabled. While the guard is `false`, the reader/writer *waits* for it to become `true`. This is implemented by associating a condition variable cv with the guard G: the reader/writer releases the lock ℓ and waits till the writer/reader *notifies* it that G could be `true`; the reader/writer then reacquires the lock and rechecks the guard. If the guard G is `true`, the reader/writer updates the values of the shared Boolean variables in parallel to indicate that it is effectively in the next code region, sends a notification signal to the writer/reader, which may be waiting for this update, and releases the lock. If the guard for a code region is always `true`, as is the case for code region TRY_1 for instance, we do not need to check its guard, and hence, do not

need a condition variable associated with its guard.

```
main() {
   boolean ncs1 := 1, try1 := 0, cs1 := 0, ncs2 := 1, try2 := 0, cs2 := 0;
   /* Definition of locks in order of predecided lock order */
   lock ℓncs1, ℓtry1, ℓcs1, ℓncs2, ℓtry2, ℓcs2;
   lock ℓcvcs1, condition variable cvcs1;
   lock ℓcvcs2, condition variable cvcs2;
   P1f() || P2f();
}
```

```
P1f() {
   while(true) {
     Execute code region NCS1;
     /* Synchronization region */
     lock(ℓncs1, ℓtry1) {
       ncs1, try1 := 0, 1;
     }
     lock(ℓcvcs2) {
       notify(cvcs2);
     }
     Execute code region TRY1;
     /* Synchronization region */
     lock(ℓcvcs1) {
       while (!Guardcs1())
          wait(cvcs1, ℓcvcs1);
     }
     Execute code region CS1;
     /* Synchronization region */
     lock(ℓncs1, ℓcs1) {
       cs1, ncs1 := 0, 1;
     }
     lock(ℓcvcs2) {
       notify(cvcs2);
 }}}

boolean Guardcs1() {
   lock(ℓtry1, ℓcs1, ℓncs2) {
     if (ncs2) {
       try1, cs1 := 0, 1;
       return(true);
     }
     else return(false);
 }}
```

```
P2f() {
   while(true) {
     Execute code region NCS2;
     /* Synchronization region */
     lock(ℓncs2, ℓtry2) {
       ncs2, try2 := 0, 1;
     }
     Execute code region TRY2;
     /* Synchronization region */
     lock(ℓcvcs2) {
       while (!Guardcs2())
          wait(cvcs2, ℓcvcs2);
     }
     Execute code region CS2;
     /* Synchronization region */
     lock(ℓncs2, ℓcs2) {
       cs2, ncs2 := 0, 1;
     }
     lock(ℓcvcs1) {
       notify(cvcs1);
 }}}

boolean Guardcs2() {
   lock(ℓncs1, ℓtry1, ℓtry2, ℓcs2) {
     if (ncs1 ∨ try1) {
       try2, cs2 := 0, 1;
       return(true);
     }
     else return(false);
 }}
```

Figure 6.7: The concurrent program $\mathcal{P}_1^f \parallel P_2^f$

The implementation of a fine-grained synchronization region is similar in essence. It differs in the use of multiple locks to control access to the shared and condition variables, thereby allowing more concurrency. As can be

seen in Fig. 6.7, $\mathcal{P}_1^f$ and $\mathcal{P}_2^f$ evaluate the guard of a code region and update the Boolean shared variables in a separate subroutine. In this subroutine, the processes first acquire all necessary locks. The acquisition of locks is in a strictly nested fashion in a predecided fixed order to prevent deadlocks. We use `lock(`$\ell_1, \ell_2, \ldots$`){...}` to denote the nested locks `lock(`ℓ_1`){lock(`ℓ_2`){...}}`, with ℓ_1 being the outermost lock variable. A synchronization region in the main body of the reader/writer process acquires the relevant lock and calls its guard-computing subroutine within a `while` loop till it returns `true`, after which it releases the lock. If the subroutine returns `false`, the process waits on the associated condition variable. Each notification signal for a condition variable, on which the other process may be waiting, is sent out by acquiring the corresponding lock. In the event that the guard of a code region is always `true`, we do not need a condition variable to control access into the code region. Hence, instead of using a separate subroutine for performing the shared variables updates, the updates are done in the main body of the synchronization region, after acquiring the necessary locks.

Both $\mathcal{P}_1^c \parallel \mathcal{P}_2^c$ and $\mathcal{P}_1^f \parallel \mathcal{P}_2^f$ are correct-by-construction and hence guaranteed to satisfy $\phi_{spec} \wedge \phi_{\mathcal{P}}$. In the following section, we provide a more detailed description of the algorithms involved in generating $\mathcal{P}_1^c \parallel \mathcal{P}_2^c$ and $\mathcal{P}_1^f \parallel \mathcal{P}_2^f$, and establish their correctness.

6.3 The Synchronization Synthesis Algorithm

In what follows, we restrict the number of sequential processes to 2 for ease of exposition. Let us review our problem definition. Given the unsynchronized skeletons of sequential processes $\mathcal{P}_1$ and $\mathcal{P}_2$, and a temporal specification ϕ_{spec} of their desired global concurrent behaviour, we wish to automatically insert synchronization code in $\mathcal{P}_1$ and $\mathcal{P}_2$, to obtain $\overline{\mathcal{P}}_1$ and $\overline{\mathcal{P}}_2$, such that $\overline{\mathcal{P}}_1 \parallel \overline{\mathcal{P}}_2 \models \phi$. We propose an automated framework to do this in two steps. The first step involves computer-aided construction of a high-level solution with synchronization actions based on guarded commands. The second step comprises a correctness-preserving, mechanical translation of the high-level synchronization actions into synchronization code based on lower-level synchronization primitives such as locks and condition variables. We describe these steps in more detail in this section.

6.3.1 Synthesis of High-Level Solution

For the first step, we primarily rely on the methodology presented in [46, 47] to: (1) synthesize a global model $\mathcal{M}$, based on $\mathcal{P}_1$ and $\mathcal{P}_2$, such that $\mathcal{M} \models \phi_{spec}$, and (2) derive the synchronization skeletons, $\mathcal{P}_1^s$ and $\mathcal{P}_2^s$ from $\mathcal{M}$. The synthesis of a global model requires a specification $\phi_{\mathcal{P}}$ of the concurrency model and control-flow of the unsynchronized process skeletons, along with the desired global behaviour ϕ_{spec}. In this work, we alleviate the user's burden of specification-writing by mainly requiring the user to provide a simple state-transition diagram representation of the unsynchronized processes. The CTL

formula $\phi_{\mathcal{P}}$, is then automatically inferred as the conjunction of the following (classes of) properties (illustrated using our reader-writer example from Fig. 6.1):

1. Initial condition:

   $\text{NCS}_1 \wedge \text{NCS}_2$

2. Each process $\mathcal{P}_k$, for $k \in \{1, 2\}$, is always in exactly one of its code regions:

   $\mathsf{AG}(\text{NCS}_k \vee \text{TRY}_k \vee \text{CS}_k) \wedge \mathsf{AG}(\text{NCS}_k \Rightarrow \neg(\text{TRY}_k \vee \text{CS}_k)) \wedge$

   $\mathsf{AG}(\text{TRY}_k \Rightarrow \neg(\text{NCS}_k \vee \text{CS}_k)) \wedge \mathsf{AG}(\text{CS}_k \Rightarrow \neg(\text{NCS}_k \vee \text{TRY}_k))$

3. At any step, only one process can make a (local) move:

   $\mathsf{AG}(\text{NCS}_1 \Rightarrow \mathsf{AX}_2\,\text{NCS}_1) \wedge \mathsf{AG}(\text{TRY}_1 \Rightarrow \mathsf{AX}_2\,\text{TRY}_1) \wedge \mathsf{AG}(\text{CS}_1 \Rightarrow \mathsf{AX}_2\,\text{CS}_1)$

   $\mathsf{AG}(\text{NCS}_2 \Rightarrow \mathsf{AX}_1\,\text{NCS}_2) \wedge \mathsf{AG}(\text{TRY}_2 \Rightarrow \mathsf{AX}_1\,\text{TRY}_2) \wedge \mathsf{AG}(\text{CS}_2 \Rightarrow \mathsf{AX}_1\,\text{CS}_2)$

4. Some process can always make a (local) move:

   $\mathsf{AG}(\mathsf{EX}_1\texttt{true} \vee \mathsf{EX}_2\texttt{true})$

   Besides the above properties, $\phi_{\mathcal{P}}$ also includes CTL formulas describing the flow of control between the code regions of each process. These properties essentially capture two types of transitions between code regions:

5. Any move that a process makes from a current code region is into a successor code region

6. Any move that a process makes from a current code region is into a successor code region *and* such a move is always possible.

Notice that property (5) simply identifies a successor code region, while property (6), in addition, states that the move into the successor code region can be made unconditionally, By default, one can assume that all transitions in the state-transition diagram of a process satisfy the stronger property (6) above. One may sometimes require a user to identify the transitions that do not satisfy property (6), and instead only satisfy property (5). However, it is typically possible to automate this classification as well. For our reader-writer example, for instance, since the code regions CS_1 and CS_2 involve a mutual exclusion property, transitions into these code regions cannot satisfy property (6). Thus, we have the following additional formulas included in $\phi_{\mathcal{P}}$ for the reader-writer example, for $k \in \{1, 2\}$: $\mathsf{AG}(\text{TRY}_k \Rightarrow \mathsf{AX}_k \, \text{CS}_k)$, $\mathsf{AG}(\text{NCS}_k \Rightarrow \mathsf{EX}_k \, \text{TRY}_k)$ and $\mathsf{AG}(\text{CS}_k \Rightarrow \mathsf{EX}_k \, \text{NCS}_k)$[4].

We refer the interested reader to [47] for further details about the steps involved in the synthesis of $\mathcal{M}$ and $\mathcal{P}_1^s$, $\mathcal{P}_2^s$. For our current purpose, it suffices to recall that the synthesis process may introduce shared synchronization variables, $\bar{x}$, and that there may be multiple labeled transitions between two code regions of a synchronization skeleton.

We would like to remark that the above high-level solution cannot be extracted by pruning the naive product graph of the local process skeletons.

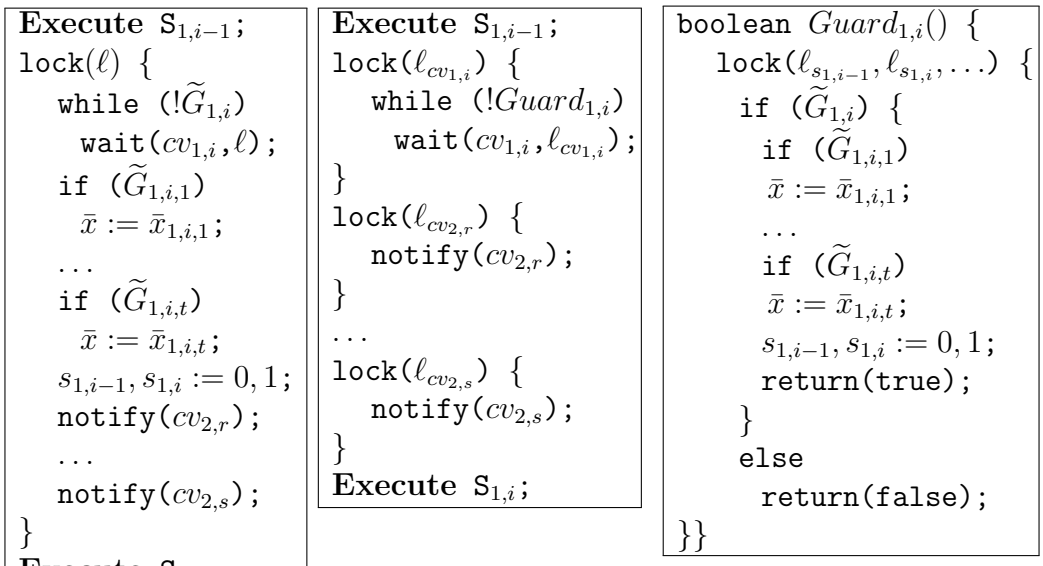
[4]Let us clarify the semantic difference between the temporal operators AX_k and EX_k. If there exists a state u in model $\mathcal{M}$ such that there is no $\mathcal{P}_k$-transition enabled in u, $\mathcal{M}, u \not\models \mathsf{EX}_k \, \mathtt{true}$, but $\mathcal{M}, u \models \mathsf{AX}_k \, \mathtt{true}$. Thus, AX_k is a weak nexttime operator, which only asserts some property of the next state *if* there exists a next state with a $\mathcal{P}_k$-transition into it. EX_k, on the other hand, is a strong nexttime operator, and also asserts the existence of a next state with a $\mathcal{P}_k$-transition into it.

This is because the product-graph-pruning approach does not introduce any additional shared synchronization variables that can help ensure fulfillment of liveness properties.

Notation: In what follows, let $\mathcal{P}_1$ have n_1 code regions, $\mathrm{S}_{1,1}, \ldots, \mathrm{S}_{1,n_1}$, and $\mathcal{P}_2$ have n_2 code regions, $\mathrm{S}_{2,1}, \ldots, \mathrm{S}_{2,n_2}$. We denote the guard and the action in the j^{th} labeled transition from $\mathrm{S}_{1,(i-1)}$ to $\mathrm{S}_{1,i}$ in $\mathcal{P}_1^s$ as $G_{1,i,j}$ and $A_{1,i,j}$, respectively. When helpful, we will explicitly write $A_{1,i,j}$ as $\bar{x} := \bar{x}_{1,i,j}$ to denote a parallel assignment to the shared synchronization variables $\bar{x}$ that results in their new valuation $\bar{x}_{1,i,j}$. We denote the disjunction over guards in all labeled transitions from $\mathrm{S}_{1,(i-1)}$ to $\mathrm{S}_{1,i}$ as $G_{1,i} = \bigvee_j G_{1,i,j}$, and refer to it as the *overall guard* of code region $\mathrm{S}_{1,i}$.

6.3.2 Synthesis of Low-level Solution

In the second step of our algorithm, we mechanically compile the guarded commands of $\mathcal{P}_1^s$ and $\mathcal{P}_2^s$ into coarse-grained or fine-grained synchronization code for $\mathcal{P}_1$ and $\mathcal{P}_2$. The resulting processes are denoted as $\mathcal{P}_1^c$, $\mathcal{P}_2^c$ (coarse-grained) or $\mathcal{P}_1^f$, $\mathcal{P}_2^f$ (fine-grained). In both cases, we first declare Boolean shared variables, $s_{1,i}$, $s_{2,r}$ to represent the code regions $\mathrm{S}_{1,i}$, $\mathrm{S}_{2,r}$ of each sequential process. We also declare locks and conditions variables for synchronization. For the program $\mathcal{P}_1^c \parallel P_2^c$, which has a coarser level of lock granularity, we declare a single lock ℓ for controlling access to shared variables and condition variables. For the program $\mathcal{P}_1^f \parallel P_2^f$ with a finer level of lock granularity, we declare separate locks $\ell_{s_{1,i}}$, $\ell_{s_{2,r}}$ and ℓ_{x_q} for controlling access to each Boolean shared variable



(a) Coarse-grained

(b) Fine-grained

Figure 6.8: Coarse, fine-grained synchronization regions between $S_{1,i-1}$, $S_{1,i}$.

$s_{1,i}$, $s_{2,k}$, and each shared synchronization variable x_q, respectively. We further define a separate lock $\ell_{cv_{1,i}}$, $\ell_{cv_{2,r}}$ for each condition variable $cv_{1,i}$, $cv_{2,r}$ to allow simultaneous processing of different condition variables.

We refer the reader to Fig. 6.8a for a generalized coarse-grained synchronization region between code regions $S_{1,i-1}$ and $S_{1,i}$. The implementation of a coarse-grained synchronization region involves acquiring the lock l and checking if the overall guard $G_{1,i}$ for entering the next code region $S_{1,i}$ is enabled. While the guard is `false`, $\mathcal{P}_1^c$ *waits* for $\mathcal{P}_2^c$ to be in an enabling code region. This is implemented by associating a condition variable $cv_{1,i}$ with the overall guard $G_{1,i}$ of $S_{1,i}$. If none of the guards of the labeled transitions is

`true`, $\mathcal{P}_1^c$ waits till $\mathcal{P}_2^c$ notifies it that $G_{1,i}$ could be `true`. $\mathcal{P}_1^c$ then reacquires the lock ℓ and rechecks $G_{1,i}$. If $G_{1,i}$ is `true`, some individual guard $G_{1,i,j}$ is `true`, and $\mathcal{P}_1^c$ performs a parallel update to the values of the $\bar{x}$ variables, as expressed in the corresponding $A_{1,i,j}$. Since we cannot modify the code region $\mathrm{S}_{1,i}$, before proceeding to $\mathrm{S}_{1,i}$, $\mathcal{P}_1^c$ also updates the values of the Boolean shared variables $s_{1,i-1}$ and $s_{1,i}$ to reflect the code region change. Finally, $\mathcal{P}_1^c$ sends a notification signal corresponding to every guard (i.e. condition variable) of $\mathcal{P}_2^c$ which may be changed to `true` by $\mathcal{P}_1^c$'s shared variables updates, and releases the lock ℓ. Note that if the overall guard for a code region is `true`, we do not need to check its guard, and hence, do not need to declare a condition variable associated with its guard.

While fine-grained locking can typically be achieved by careful definition and nesting of multiple locks, one needs to be especially cautious in the presence of condition variables for various reasons. For instance, upon execution of `wait(`c`,`ℓ_c`)` in a nested locking scheme, a process only releases the lock ℓ_c before going to sleep, while still holding all outer locks. This can potentially lead to a deadlock. The generalized fine-grained synchronization region preceding $S_{1,i}$ in $\mathcal{P}_1^f$, shown in Fig. 6.8b, circumvents these issues by utilizing a separate subroutine to evaluate the overall guard $G_{1,i}$. In this subroutine, $\mathcal{P}_1^f$ first acquires all *necessary* locks, corresponding to all shared variables accessed in the subroutine. These locks are acquired in a strictly nested fashion and in a predecided fixed order to prevent deadlocks. The subroutine then evaluates $G_{1,i}$ and returns its value to the main body of $\mathcal{P}_1^f$. If found `true`, the sub-

routine also performs an appropriate parallel update to the $\bar{x}$ variables similar to the coarse-grained case. The synchronization region in the main body of $\mathcal{P}_1^f$ acquires the relevant lock $\ell_{cv_{1,i}}$ and calls its guard-computing subroutine within a `while` loop till it returns `true`, after which it releases the lock $\ell_{cv_{1,i}}$. If the subroutine returns `false`, the process waits on the associated condition variable $cv_{1,i}$. Each notification signal for a condition variable, on which the other process may be waiting, is sent out by acquiring the corresponding lock.

We show a sketch of the overall algorithm for generating the fine-grained solution in Algo. 1. The algorithm for generating the coarse-grained solution is similar (and obviously, simpler). In Lines 2-6, we first declare a Boolean shared variable $s_{1,i}$ to represent each code region $\mathrm{S}_{1,i}$, and a corresponding lock $\ell_{s_{1,i}}$. Further, since each guard $G_{1,i,j}$ in $\mathcal{P}_1^s$ is a predicate over the code regions of $\mathcal{P}_2$ and the $\bar{x}$ variables, we transform each $G_{1,i,j}$ into $\widetilde{G}_{1,i,j}$ by mapping each $\mathrm{S}_{2,r}$ in $G_{1,i,j}$ to the corresponding Boolean shared variable $s_{2,r}$. Thus, each $\widetilde{G}_{1,i,j}$ is a predicate over the Boolean shared variables and the $\bar{x}$ variables. We similarly compute $\widetilde{G}_{1,i}$. The algorithm declares a lock $\ell_{1,i}$ and a condition variable $cv_{1,i}$, corresponding to $\mathrm{S}_{1,i}$, only if $\widetilde{G}_{1,i}$ is not equivalent to `true`.

Once all shared variable declarations for both processes are completed, the next step computes the ordered list of lock variables, corresponding to all shared variables that are accessed in the guarded commands $\widetilde{G}_{1,i,j}? \rightarrow A_{1,i,j}$ labeling the transitions from $\mathrm{S}_{1,i-1}$ to $\mathrm{S}_{1,i}$. The locks are ordered according to

a predecided fixed order.

Algorithm 1: Computation of $\mathcal{P}_1^f$, $\mathcal{P}_2^f$

Input: $\mathcal{P}_1^s$, $\mathcal{P}_2^s$, predefined lock order LO

1 **begin**

`/* Declaration of shared variables */`

2 **foreach** code region $\mathrm{S}_{1,i}$ *in* $\mathcal{P}_1^s$ **do**

3 define Boolean shared var $s_{1,i}$ and lock $\ell_{s_{1,i}}$;

4 compute $\widetilde{G}_{1,i,j}$ for each j and compute $\widetilde{G}_{1,i}$;

5 **if** $\widetilde{G}_{1,i} \not\equiv$ `true` **then** `/* i.e.,` $\widetilde{G}_{1,i} \not\equiv \bigvee_r S_{2,r}$ `*/`

6 define lock $\ell_{1,i}$ with condition var $cv_{1,i}$;

`/* Steps not shown: repeat Lines 2-6 for` $\mathcal{P}_2^s$ `*/`

`/* Computation of lock list */`

7 **foreach** $\mathrm{S}_{1,i}$ *in* $\mathcal{P}_1^s$ **do**

8 create ordered list, $LockList(\mathrm{S}_{1,i})$, with locks, corres. to all shared vars to be accessed in the synchronization region for $\mathrm{S}_{1,i}$, ordered according to LO ;

`/* Computation of notification list */`

9 **foreach** $\mathrm{S}_{1,i}$ *in* $\mathcal{P}_1^s$ **do**

10 **foreach** $cv_{2,r}$ *in* $\mathcal{P}_2^s$ **do**

11 **if** $(s_{1,i} \Rightarrow \widetilde{G}_{2,r}) \vee (s_{1,i} \wedge \bigvee_p (\bar{x} = \bar{x}_{1,i,p}) \Rightarrow \widetilde{G}_{2,r})$ **then**

12 Add $cv_{2,r}$ to $NotificationList(\mathrm{S}_{1,i})$;

`/* Construction of synchronization regions in the main body of` $\mathcal{P}_1^f$ `*/`

13 **foreach** $\mathrm{S}_{1,i}$ *in* $\mathcal{P}_1^s$ **do**

14 **if** $\widetilde{G}_{1,i} \not\equiv$ `true` **then**

15 print "`lock(`$\ell_{1,i}$`) { while (!`$Guard_{1,i}$`) wait(`$cv_{1,i}$`,`$\ell_{1,i}$`) };`";

16 **else**

17 print "`lock(`", print ordered list of lock vars from $LockList(\mathrm{S}_{1,i})$, print "`){`";

if only one labeled transition from $\mathrm{S}_{1,i-1} \xrightarrow{A_t(S_{1,i})} \mathrm{S}_{1,i}$ **then**

18 print "$\bar{x} :=$ ", print the valuation of $\bar{x}_{1,i}$, print "`;`";

19 **else**

20 **foreach** labeled transition $\mathrm{S}_{1,i-1} \xrightarrow{G_{1,i-1,j}?\to A_{1,i,j}} \mathrm{S}_{1,i}$ **do**

21 print "`if (`", print the expression $\widetilde{G}_{1,i,j}$, print "`)` $\bar{x} :=$ ", print the valuation of $\bar{x}_{1,i,j}$, print "`;`";

22 print "$s_{1,i-1}, s_{1,i} := 0, 1$`; }`";

23 **foreach** $cv_{2,k}$ *in* $NotificationList(\mathrm{S}_{1,i})$ **do**

24 print "`lock(`$\ell_{2,k}$`) { notify(`$cv_{2,k}$`); }`";

`/* Construction of synchronization subroutines of` $\mathcal{P}_1^f$ `*/`

25 **foreach** $\mathrm{S}_{1,i}$ *in* $\mathcal{P}_1^s$ **do**

26 **if** $\widetilde{G}_{1,i} \not\equiv$ `true` **then**

27 create subroutine named $Guard_{1,i}$ with a Boolean return value: "`boolean` $Guard_{1,i}$`() {`" ;

`/* Steps not shown: generate subroutine body, as shown in Fig. 7.4b, using steps similar to Lines 17-22, ending the subroutine body by printing the following */`

28 print "`return(true); } else return(false); }}`"

`/* Steps not shown: repeat Lines 7-28 for` $\mathcal{P}_2^s$ `*/`

29 **end**

We then compute the list of condition variables (Lines 9-12), on which $\mathcal{P}_2^c$ may be waiting, and whose corresponding guards may be changed from `false` to `true` by the shared variable updates done in the synchronization region for $\mathrm{S}_{1,i}$. This corresponds to the predicate $(s_{1,i} \Rightarrow \widetilde{G}_{2,r}) \vee (s_{1,i} \wedge \bigvee_p(\bar{x} = \bar{x}_{1,i,p}) \Rightarrow \widetilde{G}_{2,r})$. Recall that an arbitrary guard $\widetilde{G}_{2,r,j}$ of a labeled transition into $S_{2,r}$ is a disjunction of formulas, where each formula is either some $s_{1,i}$, or a conjunction $s_{1,i} \wedge \bar{x} = \bar{x}_{1,i,p}$ of some $s_{1,i}$ and some p^{th} valuation of the $\bar{x}$ variables. Thus, the condition variable $cv_{2,r}$ needs to be notified only if either the update $s_{1,i} := 1$ done in the synchronization region for $S_{1,i}$ makes $\widetilde{G}_{2,r,j}$ `true`, or, the update $s_{1,i} := 1$ in conjunction with some updated valuation $\bar{x}_{1,i,p}$ of the $\bar{x}$ variables makes $\widetilde{G}_{2,r,j}$ `true`.

Finally, the algorithm synthesizes synchronization regions and synchronization subroutines for both processes, of the form shown in Fig. 6.8b (Lines 13-28). Note that in the event that the guard of a code region is `true`, we do not need a condition variable to control access into the code region. Hence, instead of using a separate subroutine for performing the shared variables updates, the updates are done in the main body of the synchronization region, after acquiring the necessary locks.

We emphasize certain optimizations implemented in our compilations that potentially improve the performance of the synthesized concurrent program: (a) declaration of condition variables only when necessary, (b) inference of a $NotificationList$ for each synchronization region that avoids sending un-

necessary notification signals to the other process, and, (c) use of separate synchronization subroutines only when necessary.

The fine-grained compilation presented here illustrates the general anatomy of a fine-grained solution based on locks and condition variables. While we associate a lock with each shared variable in our solution, this need not be the case in general. The number of fine-grained locks to be used can be reduced, for example, using the approach in [49].

6.3.3 Algorithm Notes

Soundness and Completeness Let $\mathcal{M}^c$ and $\mathcal{M}^f$, be the global models[5] corresponding to $P_1^c \parallel P_2^c$ and $P_1^f \parallel P_2^f$, respectively. We have the following *Correspondence Lemmas*:

Lemma 6.3.1. *[Coarse-grained Correspondence] Given an* $\mathsf{ACTL} \setminus \mathsf{X}$ *formula* ϕ, $\mathcal{M} \models \phi \;\Rightarrow\; \mathcal{M}^c \models \phi$.

Lemma 6.3.2. *[Fine-grained Correspondence] Given an* $\mathsf{ACTL} \setminus \mathsf{X}$ *formula* ϕ, $\mathcal{M} \models \phi \;\Rightarrow\; \mathcal{M}^f \models \phi$.

The proofs of the above lemmas are based on carefully establishing *stuttering simulations* between the models, and are presented in Appendix 1[6].

[5]These global models can be constructed from the concurrent composition of P_1^c, P_2^c and P_1^f, P_2^f, respectively using the semantics of the synchronization primitives, `lock(`ℓ`)`, `wait(`c,ℓ_c`)` and `notify(`c`)`, as defined in Sec. 6.1.

[6]While we choose to restrict our attention to the preservation of $\mathsf{ACTL} \setminus \mathsf{X}$ formulas in the above lemmas, we show in Appendix 1 that the translations from $\mathcal{M}$ to $\mathcal{M}^c$ and $\mathcal{M}^f$ actually preserve all $\mathsf{ACTL}^* \setminus \mathsf{X}$ properties, as well as CTL^* properties of the form $\mathsf{A}\,\psi$ or $\mathsf{E}\,\psi$, where ψ is an $\mathsf{LTL} \setminus \mathsf{X}$ formula.

Note that the models are not stuttering bisimilar, and hence our compilations do not preserve arbitrary $\mathsf{CTL} \setminus \mathsf{X}$ properties. This is not a problem, as most global concurrency properties of interest (see Table 6.1) are expressible in $\mathsf{ACTL} \setminus \mathsf{X}$.

Theorem 6.3.3. *[Soundness]: Given unsynchronized skeletons $\mathcal{P}_1$, $\mathcal{P}_2$, and an $\mathsf{ACTL} \setminus \mathsf{X}$ formula ϕ_{spec}, if our method generates $\mathcal{P}_1^c$, $\mathcal{P}_2^c$, or, $\mathcal{P}_1^f$, $\mathcal{P}_2^f$, then $\mathcal{P}_1^c \parallel \mathcal{P}_2^c \models \phi_{spec}$ and $\mathcal{P}_1^f \parallel \mathcal{P}_2^f \models \phi_{spec}$.*

Theorem 6.3.4. *[Completeness]: Given unsynchronized skeletons $\mathcal{P}_1$, $\mathcal{P}_2$, and an $\mathsf{ACTL} \setminus \mathsf{X}$ formula ϕ_{spec}, if the temporal specifications — $\phi_{\mathcal{P}}$, describing $\mathcal{P}_1$, $\mathcal{P}_2$ and the concurrency model, and ϕ_{spec}, describing their intended global behaviour — are consistent, then our method constructs $\mathcal{P}_1^c$, $\mathcal{P}_2^c$, and $\mathcal{P}_1^f$, $\mathcal{P}_2^f$ such that $\mathcal{P}_1^c \parallel \mathcal{P}_2^c \models \phi_{spec}$ and $\mathcal{P}_1^f \parallel \mathcal{P}_2^f \models \phi_{spec}$.*

The soundness follows directly from the soundness of the synthesis of synchronization skeletons [46, 47], and from the above *Correspondence Lemmas*. The completeness follows from the completeness of the synthesis of synchronization skeletons for overall consistent specifications and from the completeness of the compilation of guarded commands to coarse-grained and fine-grained synchronization code.

Extensions. The framework presented above can be extended in a straightforward manner to the synthesis of concurrent programs based on an arbitrary (but fixed) number K of processes. The synthesis of synchronization skeletons

in the first step, based on [47], extends directly to K processes. But since this involves building a global model $\mathcal{M}$, with size exponential in K, it exhibits a state explosion problem. There has, however, been work [3, 4] on improving the scalability of the approach. These algorithms avoid building the entire global model and instead compose interacting process pairs to synthesize the synchronization skeletons. Hence, for $K > 2$ processes, we can use the more scalable synthesis algorithms to synthesize the synchronization skeletons in the first step.

The synthesis of coarse-grained synchronization can be directly extended to handle $K > 2$ processes by declaring Boolean shared variables and condition variables (when the guard is not equivalent to `true`) corresponding to the code regions of all processes, as before. The basic structure of a synthesized synchronization region remains the same. The notification lists are also computed as before and now include condition variables associated with multiple processes, in general. The synthesis of fine-grained synchronization can similarly be extended to handle $K > 2$ processes. Note that we never need to use a `notifyAll` synchronization primitive, as every condition variable is associated with a unique code region of a unique process. Further note that the synthesis of coarse or fine-grained synchronization circumvents the state-explosion problem for arbitrary K by avoiding construction/manipulation of the global model.

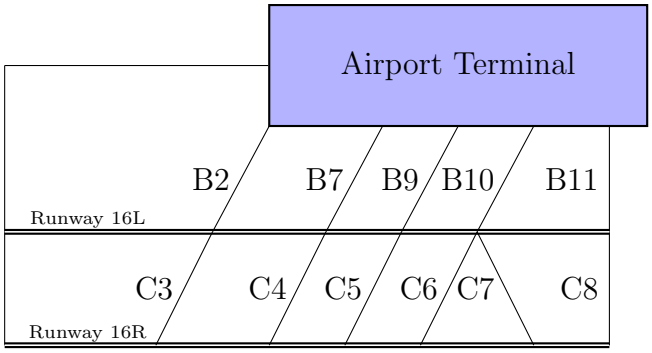


Figure 6.9: Airport ground network

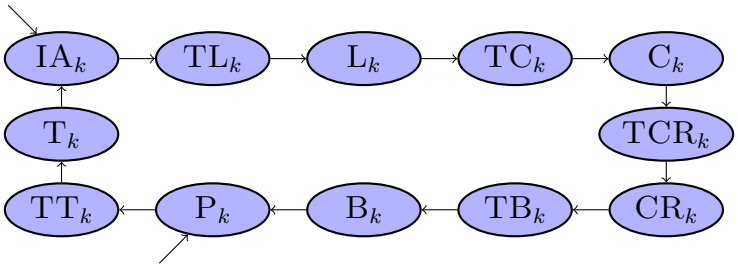


Figure 6.10: Synchronization-free skeleton for airplane process $\mathcal{P}_k$

6.4 Experiments

Example: Airport Ground Traffic Simulator (AGTS). Let us consider an example concurrent program, AGTS, based on the case study in [132], that simulates airport ground traffic network operations. The ground network model resembles the Seattle/Tacoma International Airport with two runways 16R and 16L, used for landing and takeoff, respectively, and is shown in Fig. 6.9. Each process in the AGTS program simulates the operations of a single airplane.

We assume that we are given the *synchronization-free* skeleton for the k^{th} airplane process $\mathcal{P}_k$, as shown in Fig. 6.10. This unsynchronized skeleton abstracts the actual operations of an airplane, and encodes the control flow between the *code regions* of the airplane process as a state-transition diagram, similar to Fig. 6.1. Thus, as shown in Fig. 6.10, an airplane process $\mathcal{P}_k$ is initially in state $\textbf{InAir}_k$ (arriving airplane) or $\textbf{Park}_k$ (departing airplane). An arriving airplane first attempts to land on runway 16R ($\textbf{TryLand}_k$), then lands ($\textbf{Land}_k$), attempts to exit on a C taxiway ($\textbf{TryC}_k$), takes a C taxiway ($\textbf{C}_k$), attempts to cross runway 16L ($\textbf{TryCRoss}_k$), crosses the runway ($\textbf{CRoss}_k$), attempts to take the corresponding B taxiway ($\textbf{TryB}_k$), takes the B taxiway ($\textbf{B}_k$), and finally parks. A departing airplane attempts to takeoff on runway 16L ($\textbf{TryTakeoff}_k$), takes off ($\textbf{Takeoff}_k$) and is back in air. Henceforth. we abbreviate the code region names using the corresponding bold letters. Note that, for simplicity, we restrict our attention to just one C and one B taxiway in Fig. 6.10[7].

The AGTS program must guarantee the following properties: (a) Mutual exclusion: No two airplanes can occupy the same runway or taxiway at the same time, and, an airplane taxi-ing on a C exit can cross runway 16L only if no airplane is on runway 16L at the moment, (b) Starvation-freedom[8]: An airplane that wants to land, take one of the C taxiways, cross runway 16L, or take one of the B taxiways, eventually succeeds in doing so, and (c) Priority

[7]While this restriction imposes a linear control flow on our example process $\mathcal{P}_k$, note that our framework can be extended to handle skeletons with an arbitrary branching structure.

[8]The case study in [132] focussed on safety properties.

Table 6.2: Formal specification for AGTS program for every process pair $\mathcal{P}_k, \mathcal{P}_h$

Mutual exclusion:
$\mathsf{AG}(\neg((\mathrm{L}_k \vee \mathrm{TC}_k) \wedge (\mathrm{L}_h \vee \mathrm{TC}_h)))$
$\mathsf{AG}(\neg((\mathrm{C}_k \vee \mathrm{TCR}_k) \wedge (\mathrm{C}_h \vee \mathrm{TCR}_h)))$
$\mathsf{AG}(\neg((\mathrm{CR}_k \vee \mathrm{TB}_k \vee \mathrm{T}_k) \wedge (\mathrm{CR}_h \vee \mathrm{TB}_h \vee \mathrm{T}_h)))$
$\mathsf{AG}(\neg(\mathrm{B}_k \wedge \mathrm{B}_h))$

Starvation-freedom:
$\mathsf{AG}(\mathrm{TL}_k \Rightarrow \mathsf{AF}\,\mathrm{L}_h)$ $\quad$ $\mathsf{AG}(\mathrm{TCR}_k \Rightarrow \mathsf{AF}\,\mathrm{CR}_h)$
$\mathsf{AG}(\mathrm{TC}_k \Rightarrow \mathsf{AF}\,\mathrm{C}_h)$ $\quad$ $\mathsf{AG}(\mathrm{TB}_k \Rightarrow \mathsf{AF}\,\mathrm{B}_h)$

Priority of arriving plane over departing plane:
$\mathsf{AG}((\mathrm{TT}_k \wedge (\mathrm{TCR}_h \vee \mathrm{CR}_h \vee \mathrm{TB}_h)) \Rightarrow \mathsf{A}[\mathrm{TT}_k\,\mathsf{U}\,\mathrm{B}_h])$

Starvation-freedom for departing plane, provided arriving plane is not trying to cross or crossing runway 16L:
$\mathsf{AG}(\mathrm{TT}_k \Rightarrow \mathsf{AF}(\mathrm{T}_k \vee (\mathrm{TCR}_h \vee \mathrm{CR}_h \vee \mathrm{TB}_h)))$.

of arriving airplanes: An airplane that wants to take off can do so only if no airplane wants to cross or is crossing runway 16L at the moment. We assume that we are given a formal specification capturing the desired set of properties (see Table 6.2).

Experimental Results. We have implemented a prototype synthesis tool [48] in Perl, which automatically compiles synchronization skeletons into concurrent Java programs based on both coarse-grained and fine-grained synchronization. We used the tool successfully to synthesize synchronization code for the above example AGTS program, with two airplane processes, and for several configurations of K-process mutual exclusion, readers-writers, dining philosophers, etc..

Our experiments were run on a quad-core 3.4GHz machine with 4GB of RAM. The running time of the tool for generating these small examples was a few milliseconds. We present the normalized running times for some of the

Table 6.3: Experimental results

Program	Granularity	Norm. Run. Time
2-plane AGTS	Coarse	1
	Fine	1.47
1-Reader, 1-Writer	Coarse	1
	Fine	0.79
2-process Mutex	Coarse	1
	Fine	1.08
2-Readers, 3-Writers	Coarse	1
	Fine	1.14

generated examples in Table 6.3. Observe that the coarse-grained synchronization version often outperforms the fine-grained synchronization version. In particular, it suffers significantly in the 2-plane AGTS example due to excessive locking overhead[9].

[9]For the 2-plane AGTS example, we invoked an additional optimization in the compilation: if the overall guard $\widetilde{G}_{1,i}$ of a code region $S_{1,i}$ includes more than half the code regions of $\mathcal{P}_2$, instead of using $\widetilde{G}_{1,i}$ in the coarse-grained and fine-grained compilations, we use the negation of its conjunction. This helps minimize the number of shared variables accessed in each synchronization region, and the corresponding locking overhead in the fine-grained synchronization regions.

Chapter 7

Generalized Synchronization Synthesis

Overview. In this chapter, we present a framework that takes a shared-memory concurrent program composed of unsynchronized processes, along with a temporal specification of their global concurrent behaviour, and automatically generates a concurrent program with synchronization ensuring correct global behaviour. Our methodology supports finite-state concurrent programs composed of processes that may have local and shared variables, may be straight-line or branching programs, may be ongoing or terminating, and may have program-initialized or user-initialized variables. The specification language is an extension of propositional CTL that enables easy specification of safety and liveness properties over control and data variables. The framework also supports synthesis of synchronization at different levels of abstraction and granularity.

In Chapter 6, we focussed on synthesizing synchronization code for abstract sequential processes called skeletons. This abstraction is particularly helpful when one is exclusively interested in properties describing the desired control-flow and concurrent behaviour of a concurrent program. However, this abstrac-

tion, which suppresses all sequential computational details that are irrelevant for synchronization, is restrictive, when one is interested in properties such as $\mathsf{AG}\,(v_1 = \mu \Rightarrow \mathsf{AF}(v_2 = \mu + 1))$, over the values of the data variables in a concurrent program. In this chapter, we generalize both the concurrent program model as well as the specification language of Chapter 6 to enable synthesis of synchronization in real-world programs, given temporal logic properties over both program locations and data variables.

7.1 Formal Framework

In this section, we first establish a vocabulary $\mathbf{L}$ of symbols, and then define our model of concurrent programs and specification language, based on $\mathbf{L}$.

7.1.1 A vocabulary L

Symbols of L: We fix a vocabulary $\mathbf{L}$ that includes a set $\mathbf{L}^{\mathbb{V}}$ of variable symbols (denoted v, v_1 etc.), a set $\mathbf{L}^{\mathbb{F}}$ of function symbols (denoted f, f_1 etc.), a set $\mathbf{L}^{\mathbb{B}}$ of predicate symbols (denoted B, B_1 etc.), and a non-empty set $\mathbf{L}^{\mathbb{S}}$ of *sorts*. $\mathbf{L}^{\mathbb{S}}$ contains the special sort `bool`, along with the special sort `location`. Each variable v has associated with it a sort in $\mathbf{L}^{\mathbb{S}}$, denoted $sort(v)$. Each function symbol f has an associated arity and a sort: $sort(f)$ for an r-ary function symbol is an $r+1$-tuple $< \sigma_1, \ldots, \sigma_r, \sigma >$ of sorts in $\mathbf{L}^{\mathbb{S}}$, specifying the sorts of both the domain and range of f. Each predicate symbol B also has an associated arity and sort: $sort(B)$ for an r-ary predicate symbol is an r-tuple $< \sigma_1, \ldots, \sigma_r >$ of sorts in $\mathbf{L}^{\mathbb{S}}$. Constant symbols (denoted c, c_1 etc.)

are identified as the 0-ary function symbols, with each constant symbol c associated with a sort, denoted $sort(c)$, in $\mathbf{L}^{\mathbb{S}}$. Proposition symbols are identified as the 0-ary predicate symbols. The vocabulary $\mathbf{L}$ also includes the distinguished equality predicate symbol $=$, used for comparing elements of the same sort.

Syntax of L-terms and L-atoms: Given any set of variables $V \subseteq \mathbf{L}^{\mathbb{V}}$, we inductively construct the set of $\mathbf{L}$-terms and $\mathbf{L}$-atoms over V, using sorted symbols, as follows:

- Every variable of sort σ is a term of sort σ.
- If f is a function symbol of sort $< \sigma_1, \ldots, \sigma_r, \sigma >$, and t_j is a term of sort σ_j for $j \in [1, r]$, then $f(t_1, \ldots, t_r)$ is a term of sort σ. In particular, every constant of sort σ is a term of sort σ.
- If B is a predicate symbol of sort $< \sigma_1, \ldots, \sigma_r >$, and t_j is a term of sort σ_j for $j \in [1, r]$, then $B(t_1, \ldots, t_r)$ is an atom. In particular, every proposition symbol is an atom.
- If t_1, t_2 are terms of the same sort, $t_1 = t_2$ is an atom.

Semantics of L-terms and L-atoms: Given any set of variables $V \subseteq \mathbf{L}^{\mathbb{V}}$, an *interpretation* I of symbols of $\mathbf{L}$, and $\mathbf{L}$-terms and $\mathbf{L}$-atoms over V is a map satisfying the following:

- Every sort $\sigma \in \mathbf{L}^{\mathbb{S}}$ is mapped to a nonempty domain D_σ. In particular, the sort `bool` is mapped to the Boolean domain $D^{\texttt{bool}} : \{\texttt{true}, \texttt{false}\}$, and the sort `location` is mapped to a domain of *control locations* in a program.

- Every variable symbol v of sort σ is mapped to an element v^I in D_σ.

- Every function symbol f, of sort $< \sigma_1, \ldots, \sigma_r, \sigma >$ is mapped to a function $f^I : D_{\sigma_1} \times \ldots D_{\sigma_r} \to D_\sigma$. In particular, every constant symbol c of sort σ is mapped to an element $c^I \in D_\sigma$.

- Every predicate symbol B of sort $< \sigma_1 \ldots \sigma_r >$ is mapped to a function $D_{\sigma_1} \times \ldots D_{\sigma_r} \to D^{\texttt{bool}}$. Every proposition symbol is mapped to an element of $D^{\texttt{bool}}$.

Given an interpretation I as defined above, the valuation $val^I[t]$ of an $\mathbf{L}$-term t and the valuation $val^I[G]$ of an $\mathbf{L}$-atom G are defined as follows:

- For a term t which is a variable v, the valuation is v^I.

- For a term $f(t_1, \ldots, t_r)$, the valuation $val^I[f(t_1, \ldots, t_r)]$ equals the valuation $f^I(val^I[t_1], \ldots, val^I[t_r])$.

- For an atom $G(t_1, \ldots, t_r)$, the valuation $val^I[G(t_1, \ldots, t_r)] = \texttt{true}$ iff $G^I(val^I[t_1], \ldots, val^I[t_r]) = \texttt{true}$.

- For an atom $t_1 = t_2$, $val^I[t_1 = t_2] = \texttt{true}$ iff $val^I[t_1] = val^I[t_2]$.

In the rest of this chapter, we assume that the interpretation of constant, function and predicate symbols in **L** is known and fixed. We further assume that the interpretation of sort symbols to specific domains is known and fixed. With some abuse of notation, we shall denote the interpretation of all constant, function and predicate symbols simply by the symbol name, and identify sorts with their domains. Examples of some constant, function and predicate symbols that may be included in **L** are: constant symbols $0, 1, 2$, function symbols $+, -$, and predicate symbols $<, >$ over the integers, function symbols $\vee, \neg$ over `bool`, the constant symbol φ (empty list), function symbol $\bullet$ (appending lists) and predicate symbol *null* (emptiness test) over lists, etc.. Finally, when the interpretation is obvious from the context, we denote the valuations $val^I[t]$, $val^I[G]$ of terms t and atoms G simply as $val[t]$, $val[G]$, respectively.

7.1.2 Concurrent Program Model

In this chapter, we consider a (shared-memory) concurrent program to be an asynchronous composition of a non-empty, finite set of processes, equipped with a finite set of program variables that range over finite domains. We assume a simple concurrent programming language, as shown in Fig. 7.1, with assignment, condition test, unconditional goto, sequential and parallel composition, and the synchronization primitive - conditional critical region (CCR) [63,66]. A concurrent program $\mathcal{P}$ is written using the concurrent programming language, in conjunction with **L**-terms and **L**-atoms. We assume that the sets

of (data and control) variables, functions and predicates available for writing $\mathcal{P}$ are each finite subsets of $\mathbf{L}^{\mathbb{V}}$, $\mathbf{L}^{\mathbb{F}}$ and $\mathbf{L}^{\mathbb{B}}$, respectively.

$\langle pgm \rangle$	::=	$\langle vardecl \rangle \, \langle asyncprocs \rangle$
$\langle vardecl \rangle$	::=	$v : \langle domain \rangle$; | $v : \langle domain \rangle$ `with` $v = v_{init}$;
		| $\langle vardecl \rangle \, \langle vardecl \rangle$
$\langle asyncprocs \rangle$	::=	$\langle proc \rangle \parallel \langle asyncprocs \rangle \mid \langle proc \rangle$
$\langle proc \rangle$	::=	$\langle localvardecl \rangle \, \langle stmtseq \rangle$
$\langle localvardecl \rangle$	::=	$\langle vardecl \rangle$
$\langle stmtseq \rangle$	::=	$\langle labstmt \rangle$; $\langle stmtseq \rangle$
$\langle labstmt \rangle$	::=	$\lambda : \langle atomicstmt \rangle$
$\langle atomicstmt \rangle$	::=	$\langle assignment \rangle \mid \langle conditiontest \rangle \mid \langle goto \rangle \mid \langle CCR \rangle$
$\langle assignment \rangle$	::=	$v_1, \ldots, v_m := t_1, \ldots, t_m$
$\langle conditiontest \rangle$	::=	`if` (G) $\lambda_{\texttt{if}}$ `else` $\lambda_{\texttt{else}}$
$\langle goto \rangle$	::=	`goto` λ
$\langle CCR \rangle$	::=	$G? \rightarrow \langle atomicstmtseq \rangle$

Figure 7.1: Concurrent program syntax

Thus, a concurrent program $\mathcal{P}$ consists of a variable declaration, followed by an asynchronous parallel composition, $\mathcal{P}_1 \| \ldots \| \mathcal{P}_K$, of say K processes, with $K > 0$. The variable declaration consists of a finite sequence of declaration statements, specifying the set X of shared data variables, their domains, and possibly initializing them to specific values. For example, the declaration statement, $v_1 : \{0, 1, 2, 3\}$ `with` $v_1 = 0$, declares a variable v_1 with (a finite integer) domain $\{0, 1, 2, 3\}$, and initializes the variable to the value 0. The initial value of any uninitialized variable is assumed to be a user/environment input from the domain of the variable.

A process $\mathcal{P}_k$ consists of a declaration of a set Y_k of local data variables (similar to the declaration of shared data variables in $\mathcal{P}$), and a finite sequence

of labeled, *atomic* statements. We denote the unique atomic statement at location λ as $stmt(\lambda)$. The set of data variables Var_k accessible by P_k is given by $X \cup Y_k$. The set of labels or *locations* of P_k is denoted $\mathcal{L}_k = \{\lambda_k^0, \ldots, \lambda_k^{n_k}\}$, with λ_k^0 being a designated start location. Unless specified otherwise[1], an atomic statement is an assignment, condition test, unconditional goto, or CCR. An assignment statement is a parallel assignment of **L**-terms $t_1, \ldots, t_m$, over Var_k, to the data variables $v_1, \ldots, v_m$ in Var_k. Upon completion, an assignment statement at λ_k^i transfers control to the next location λ^{i+1}. A condition test, consists of an **L**-atom G over Var_k, and a pair of locations $\lambda_{\texttt{if}}, \lambda_{\texttt{else}}$ in $\mathcal{L}_k$ to transfer control to if G evaluates to `true`, `false`, respectively. The statement `goto` λ is a transfer of control to location $\lambda \in \mathcal{L}_k$. A CCR is a guarded statement block, where the enabling guard G is an **L**-atom over Var_k and the statement block is a sequence of assignment, conditional and goto statements. The guard G is evaluated atomically and if found to be `true`, the corresponding statement block is executed atomically, and control is transferred to the next location. If G is found to be `false`, the process *waits* at the same location till G evaluates to `true`. An unsynchronized process does not contain CCRs.

We model the asynchronous composition of concurrent processes by the nondeterministic interleaving of their atomic instructions. Hence, at each step of the computation, some process, with an enabled transition, is nondeterministically selected to be executed next by a scheduler. The set of program

[1] A user may define an atomic statement (block) as a sequence of assignment, conditional and goto statements

variables is denoted $V = Loc \cup Var$, where $Loc = \{loc_1, \ldots, loc_K\}$ is the set of control variables and $Var = Var_1 \cup \ldots \cup Var_K$ is the set of data variables. The semantics of the concurrent program $\mathcal{P}$ is given by a transition system (S, S_0, R), where S is a set of states, $S_0 \subseteq S$ is a set of initial states and $R \subseteq S \times S$ is the transition relation. Each state $s \in S$ is a valuation of the program variables in V. We denote the value of variable v in state s as $val^s[v]$, and the corresponding value of a term t and an atom G in state s as $val^s[t]$ and $val^s[G]$, respectively. $val^s[t]$ and $val^s[G]$ are defined inductively as in Sec. 7.1.1. The domain of each control variable $loc_k \in V$ is the set of locations $\mathcal{L}_k$, and the domain of each data variable is determined from its declaration. The set of initial states S_0 corresponds to all states s with $val^s[loc_k] = \lambda_k^0$ for all $k \in [1, K]$, and $val^s[v] = v_{init}$, for every data variable v initialized in its declaration to some constant v_{init}. There exists a transition from state s to s' in R, with $val^s[loc_k] = \lambda_k$, $val^{s'}[loc_k] = \lambda'_k$ and $val^{s'}[loc_j] = val^s[loc_j]$ for all $j \neq k$, iff there exists a corresponding *local move* in process $\mathcal{P}_k$ involving $stmt(\lambda_k)$, such that:

1. $stmt(\lambda_k)$ is the assignment instruction: $v_1, \ldots, v_m := t_1, \ldots, t_m$, for each variable v_i with $i \in [1, m]$: $val^{s'}[v_i] = val^s[t_i]$, for all other data variables v: $val^{s'}[v] = val^s[v]$, and λ'_k is the next location in $\mathcal{P}_k$ after λ_k, or,

2. $stmt(\lambda_k)$ is the condition test: $\texttt{if}\ (G)\ \lambda_{\texttt{if}}\ \texttt{else}\ \ell_{\texttt{else}}$, the valuation of all data variables in s' is the same as that in s, and either $val^s[G]$ is `true` and $\lambda'_k = \lambda_{if}$, or $val^s[G]$ is `false` and $\lambda'_k = \lambda_{else}$, or,

3. $stmt(\lambda_k)$ is `goto` λ, the valuation of all data variables in s' is the same as that in s, and $\lambda'_k = \lambda$, or,

4. $stmt(\lambda_k)$ is the CCR $G? \rightarrow \langle atomicstmtseq \rangle$, $val^s[G]$ is `true`, the valuation of all data variables in s' correspond to the atomic execution of $\langle atomicstmtseq \rangle$ from state s, and λ'_k is the next location in $\mathcal{P}_k$ after λ_k.

We assume that R is total. For terminating processes $\mathcal{P}_k$, we assume that $\mathcal{P}_K$ ends with a special instruction, $halt$: `goto` $halt$.

7.1.3 Specification Language

Our specification language, **L**CTL, is an extension of propositional CTL, with formulas composed from **L**-atoms. While one can use propositional CTL for specifying properties of finite-state programs, **L**CTL enables more natural specification of properties of concurrent programs communicating via typed shared variables. We describe the syntax and semantics of this language below.

Syntax: Given a set of variables $V \subseteq \mathbf{L}^{\mathbb{V}}$, we inductively construct the set of (**L**CTL) formulas over V, using **L**-atoms, in conjunction with the propositional operators $\neg$, $\wedge$ and the temporal operators $\mathsf{A}, \mathsf{E}, \mathsf{X}, \mathsf{U}$, along with the process-indexed next-time operator X_k:

- Every **L**-atom over V is a formula.

- If ϕ_1, ϕ_2 are formulas, then so are $\neg\phi_1$ and $\phi_1 \wedge \phi_2$.

- If ϕ_1, ϕ_2 are formulas, then so are $\mathsf{EX}\,\phi_1$, $\mathsf{EX}_k\,\phi_1$, $\mathsf{A}[\phi_1\,\mathsf{U}\,\phi_2]$ and $\mathsf{E}[\phi_1\,\mathsf{U}\,\phi_2]$.

We use the following standard abbreviations: $\phi_1 \vee \phi_2$ for $\neg(\neg\phi_1 \wedge \neg\phi_2)$, $\phi_1 \rightarrow \phi_2$ for $\neg\phi_1 \vee \phi_2$, $\phi_1 \leftrightarrow \phi_2$ for $(\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)$, $\mathsf{AX}\,\phi$ for $\neg\mathsf{EX}\,\neg\phi$, $\mathsf{AX}_k\,\phi$ for $\neg\mathsf{EX}_k\,\neg\phi$, $\mathsf{AF}\,\phi$ for $\mathsf{A}[\mathtt{true}\,\mathsf{U}\,\phi]$, $\mathsf{EF}\,\phi$ for $\mathsf{E}[\mathtt{true}\,\mathsf{U}\,\phi]$, $\mathsf{EG}\,\phi$ for $\neg\mathsf{AF}\,\neg\phi$, and $\mathsf{AG}\,\phi$ for $\neg\mathsf{EF}\,\neg\phi$.

Semantics: **L**CTL formulas over a set of variables V are interpreted over models of the form $\mathcal{M} = (S, R, L)$, where S is a set of states and R is a a total, multi-process, binary relation $R = \cup_k R_k$ over S, composed of the transitions R_k of each process $\mathcal{P}_k$. L is a labeling function that assigns to each state $s \in S$ a valuation of all variables in V. The value of a term t in a state $s \in S$ of $\mathcal{M}$ is denoted as $val^{(\mathcal{M},s)}[t]$, and is defined inductively as in Sec. 7.1.1. A path in $\mathcal{M}$ is a sequence $\pi = (s_0, s_1, \ldots)$ of states such that $(s_j, s_{j+1}) \in R$, for all $j \geq 0$. We denote the j^{th} state in π as π_j.

The satisfiability of an **L**CTL formula in a state s of $\mathcal{M}$ can be defined as follows:

- $\mathcal{M}, s \models G(t_1, \ldots, t_m)$ iff $G(val^{(\mathcal{M},s)}[t_1], \ldots, val^{(\mathcal{M},s)}[t_m]) = \mathtt{true}$.

- $\mathcal{M}, s \models (t_1 = t_2)$ iff $val^{(\mathcal{M},s)}[t_1] = val^{(\mathcal{M},s)}[t_2]$.

- $\mathcal{M}, s \models \neg\phi$ iff it is not the case that $\mathcal{M}, s \models \phi$.

- $\mathcal{M}, s \models \phi_1 \wedge \phi_2$ iff $\mathcal{M}, s \models \phi_1$ and $\mathcal{M}, s \models \phi_2$.

- $\mathcal{M}, s \models \mathsf{EX}\,\phi$ iff for some s_1 such that $(s, s_1) \in R$, $\mathcal{M}, s_1 \models \phi$.

- $\mathcal{M}, s \models \mathsf{EX}_k\,\phi$ iff for some s_1 such that $(s, s_1) \in R_k$, $\mathcal{M}, s_1 \models \phi$.

- $\mathcal{M}, s \models \mathsf{A}[\phi_1\,\mathsf{U}\,\phi_2]$ iff for all paths π starting at s, there exists some j such that $\mathcal{M}, \pi_j \models \phi_2$ and for all $i < j$, $\mathcal{M}, \pi_i \models \phi_1$.

- $\mathcal{M}, s \models \mathsf{E}[\phi_1\,\mathsf{U}\,\phi_2]$ iff there exists a path π starting at s such that for some j, $\mathcal{M}, \pi_j \models \phi_2$ and for all $i < j$, $\mathcal{M}, \pi_i \models \phi_1$.

We say $\mathcal{M} \models \phi$, if for every s in a set of initial states in $\mathcal{M}$, $\mathcal{M}, s \models \phi$.

Programs as Models: A program $\mathcal{P} = (S, S_0, R)$ can be viewed as a model $\mathcal{M} = (S, R, L)$, with the same set of states and transitions as $\mathcal{P}$, and the identity labeling function L that maps a state to itself. Given an $\mathbf{L}CTL$ specification ϕ, we say $\mathcal{P} \models \phi$ iff for each state $s \in S_0$, $\mathcal{M}, s \models \phi$.

Example: We refer the reader to Fig. 7.2 for an example concurrent program $\mathcal{P}$ and specification ϕ_{spec} in the above program model and specification language, respectively. The symbols $\dot{+}$ and $\dot{-}$ respectively denote addition and subtraction modulo 2. Observe that $\mathcal{P}$ does not contain any CCRs and $\mathcal{P} \not\models \phi_{spec}$.

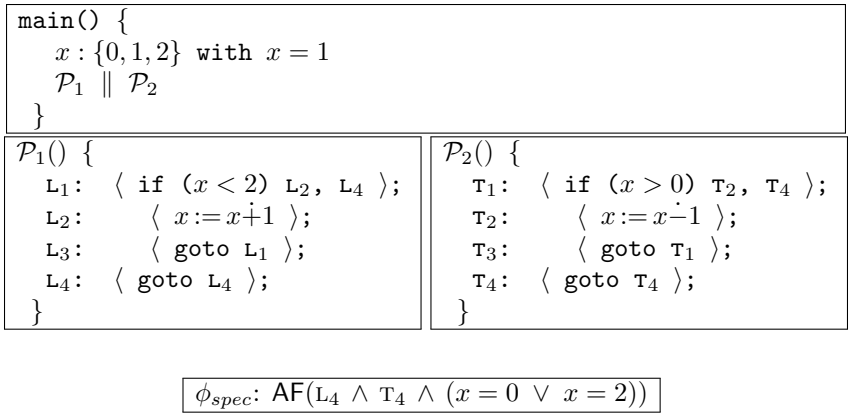


Figure 7.2: Example program $\mathcal{P}$ and specification ϕ_{spec}

7.2 The Basic Synchronization Synthesis Algorithm

In this section, for ease of exposition, we assume a simpler program model than the one described in Sec. 7.1.2. We restrict the number of concurrent processes K to 2. We assume that *all* data variables are initialized in the program to specific values from their respective domains. We further assume that all program variables, including control variables, are shared variables. We explain our basic algorithmic framework with these assumptions, and later describe extensions to handle the general program model in Sec. 7.3.

Let us present our problem definition. Given a concurrent program $\mathcal{P}$, composed of unsynchronized processes $\mathcal{P}_1$, $\mathcal{P}_2$, and an **L**CTL specification ϕ_{spec} of their desired global concurrent behaviour, we wish to automatically generate synchronized processes $\mathcal{P}_1^s$, $\mathcal{P}_2^s$, such that the resulting concurrent program $\mathcal{P}^s \models \phi_{spec}$. In particular, we wish to obtain synchronization in the form of CCRs, with each atomic statement of $\mathcal{P}_1$, $\mathcal{P}_2$ enclosed in a CCR;

the goal is to synthesize the guard for each CCR, along with any necessary assignments to synchronization variables to be performed within the CCR. We propose an automated framework to do this in several steps.

1. Formulate an **L**CTL formula $\phi_{\mathcal{P}}$ to specify the concurrency model and the operational semantics of the concurrent program $\mathcal{P}$.

2. Construct a tableau T_ϕ for the formula ϕ given by $\phi_{\mathcal{P}} \wedge \phi_{spec}$. If T_ϕ is empty, declare specification as inconsistent and halt.

3. If T_ϕ is non-empty, extract a model $\mathcal{M}$ for ϕ from it.

4. Decompose $\mathcal{M}$ to obtain CCRs to synchronize each process.

In what follows, we describe these steps in more detail. We emphasize that while these steps resemble the ones presented in Sec. 6.3, the technical details are different due to differences in the concurrent program models and specification languages.

7.2.1 Formulation of ϕ_P

As mentioned in Sec. 6.3.1, the early synthesis work in [47] requires a complete specification, which includes a temporal description $\phi_{\mathcal{P}}$ of the concurrency and operational semantics of the unsynchronized concurrent program $\mathcal{P}$, along with its desired global behaviour ϕ_{spec}. Similar to Sec. 6.3.1, we automatically infer an **L**CTL formula for $\phi_{\mathcal{P}}$ to help mitigate the user's burden of specification-writing. Let $Var = \{v_1, \ldots, v_h\}$ be the set of data variables. $\phi_{\mathcal{P}}$ is then formulated as the conjunction of the following (classes of) properties:

1. Initial condition:

   $val[loc_1] = \lambda_1^0 \ \wedge \ val[loc_2] = \lambda_2^0 \ \wedge \ \bigwedge_{v \in Var} \ val[v] = v_{init}$.

2. At any step, only one process can make a (local) move:

   $\mathsf{AG} \bigwedge_{i=1}^{n_1} (val[loc_1] = \lambda_1^i \ \Rightarrow \ \mathsf{AX}_2 \, val[loc_1] = \lambda_1^i) \quad \wedge$

   $\mathsf{AG} \bigwedge_{i=1}^{n_2} (val[loc_2] = \lambda_2^i \ \Rightarrow \ \mathsf{AX}_1 \, val[loc_2] = \lambda_2^i)$.

3. Some process can always make a (local) move:

   $\mathsf{AG}(\mathsf{EX}_1 \, \mathtt{true} \ \vee \ \mathsf{EX}_2 \, \mathtt{true})$.

4. A statement $\lambda_k^i : \{v_1, \ldots, v_m\} := \{t_1, \ldots, t_m\}$ in P_k is formulated as:

   $\mathsf{AG}((val[loc_k] = \lambda_k^i \ \wedge \ \bigwedge_{j=1}^{h} \ val[v_j] = \mu_j) \ \Rightarrow$

   $\mathsf{AX}_k \, (val[loc_k] = \lambda_k^{i+1} \ \wedge \ \bigwedge_{j=1}^{m} \ val[v_j] = val[t_j] \wedge$

   $\bigwedge_{v_j \in Var \setminus \{v_1, \ldots, v_m\}} \ val[v_j] = \mu_j))$.

5. A statement λ_k: $\mathtt{if}$ (G) $\lambda_{\mathtt{if}}$, $\lambda_{\mathtt{else}}$ in $\mathcal{P}_k$ is formulated as:

   $\mathsf{AG}((val[loc_k] = \lambda_k \ \wedge \ val[G] = \mathtt{true}) \ \Rightarrow \ \mathsf{AX}_k \, val[loc_k] = \lambda_{\mathtt{if}}) \ \wedge$

   $\mathsf{AG}((val[loc_k] = \ell_k \ \wedge \ val[G] = \mathtt{false}) \ \Rightarrow \ \mathsf{AX}_k \, val[loc_k] = \lambda_{\mathtt{else}})$.

6. A statement λ_k: $\mathtt{goto}$ λ in $\mathcal{P}_k$ is formulated as:

   $\mathsf{AG}(val[loc_k] = \lambda_k \ \Rightarrow \ \mathsf{AX}_k \, val[loc_k] = \lambda)$

7.2.2 Construction of T_ϕ

We assume the ability to evaluate **L**-atoms and **L**-terms over the set V of program variables. Note that since we restrict ourselves to a finite subset of the symbols in **L**, this is a reasonable assumption. Let us further assume that the formula $\phi = \phi_{\mathcal{P}} \wedge \phi_{spec}$ is in a form in which only atoms appear negated.

An *elementary* formula of **L**CTL is an atom, negation of an atom or the formulas beginning with AX_k or EX_k (we do not explicitly consider formulas beginning with AX or EX since $\mathsf{AX}\,\psi = \bigwedge_k \mathsf{AX}_k\,\psi$, and $\mathsf{EX}\,\psi = \bigvee_k \mathsf{EX}_k\,\psi$. All other formulas are nonelementary. Every nonelementary formula is either a conjunctive formula $\alpha \equiv \alpha_1 \wedge \alpha_2$ or a disjunctive formula $\beta \equiv \beta_1 \vee \beta_2$. For example, $\psi_1 \wedge \psi_2$, $\mathsf{AG}\,(\psi) \equiv \psi \wedge \mathsf{AXAG}\,\psi$ are α formulas, and $\psi_1 \vee \psi_2$, $\mathsf{AF}\,(\psi) \equiv \psi \vee \mathsf{AXAF}\,\psi$ are β formulas.

The tableau T_ϕ for the formula ϕ is a finite, rooted, directed AND/OR graph with nodes labeled with formulas such that when a node B is viewed as a state in a suitable structure, $B \models \psi$ for all formulas $\psi \in B$. The construction for T_ϕ is similar to the tableau-construction for propositional CTL in [47], while accounting for the presence of **L**-atoms over V in the nodes of T_ϕ. Besides composite **L**-atoms and **L**CTL formulas, each node of T_ϕ is labeled with simple atoms of the type $loc = \lambda$ and $v = \mu$ identifying the values of the control and data variables in each node. Two OR-nodes B_1 and B_2 are identified as being equivalent if B_1, B_2 are labeled with the same simple atoms, and the conjunction of all the formulas in B_1 is valid iff the conjunction of all the formulas in B_2 is valid. Equivalence of AND-nodes can be similarly defined. We briefly summarize the tableau construction first, before explaining the individual steps in more detail.

1. Initially, let the root node of T_ϕ be an OR-node labeled with ϕ.

2. If all nodes in T_ϕ have successors, go to the next step. Otherwise, pick a

node B without successors. Create appropriately labeled successors of B such that: if B is an OR-node, the formulas in B are valid iff the formulas in some (AND-) successor node are valid, and if B is an AND-node, the formulas in B are valid iff the formulas in all (OR-) successor nodes are valid. Merge all equivalent AND-nodes and equivalent OR-nodes. Repeat this step.

3. Delete all *inconsistent* nodes in the tableau from the previous step to obtain the final T_ϕ.

Successors of OR-nodes: To construct the set of AND-node successors of an OR-node B, first build a temporary tree with labeled nodes rooted at B, repeating the following step until all leaf nodes are only labeled with elementary formulas. For any leaf node C labeled with a non-elementary formula ψ: if ψ is an α formula, add a single child node, labeled $C \setminus \{\psi\} \cup \{\alpha_1, \alpha_2\}$, to C, and if ψ is a β formula, add two child nodes, labeled $C \setminus \{\psi\} \cup \{\beta_1\}$ and $C \setminus \{\psi\} \cup \{\beta_2\}$, to C. Once the temporary tree is built, create an AND-node successor D for B, corresponding to each leaf node in the tree, labeled with the set of all formulas appearing in the path to the leaf node from the root of the tree. If there exists an atom of the form $v = t$ in D, where t is an **L**-term, and the valuation of t in D is μ, replace the atom $v = t$ by the simple atom $v = \mu$.

Successors of AND-nodes: To construct the set of OR-node successors of

an AND-node B, create an OR-node labeled with $\{\psi\}$ for each $\mathsf{EX}_k\,\psi$ formula in B and label the transition to the OR-node with k. Furthermore, label each such OR-node D (with an k-labeled transition into D) with $\bigcup_j \psi_j$ for each $\mathsf{AX}_k\,\psi_j$ formula in B. If there exists an atom of the form $v = t$ in D, where t is an **L**-term, and the valuation of t in D is μ, replace the atom $v = t$ by the simple atom $v = \mu$. Note that the requirement that some process can always move ensures that there will be some successor for every AND-node.

Deletion rules: All nodes in the tableau that do not meet all criteria for a tableau for ϕ are identified as inconsistent and deleted as follows:

1. Delete any node B which is internally inconsistent, i.e., the conjunction of all non-temporal elementary formulas in B evaluates to `false`.

2. Delete any node all of whose original successors have been deleted.

3. Delete any node B such that $\mathsf{E}[\psi_1\,\mathsf{U}\,\psi_2] \in B$, and there does not exist some path to an AND-node D from B with $\psi_2 \in D$, and $\psi_1 \in C$ for all AND-nodes C in the path.

4. Delete any node B such that $\mathsf{A}[\psi_1\,\mathsf{U}\,\psi_2] \in B$, and there does not exist a full sub-DAG [2], rooted at B, such that for all its frontier nodes D , $\psi_2 \in D$ and for all its non-frontier nodes C, $\psi_1 \in C$.

[2] A full sub-DAG T' is a directed acyclic sub-graph of a tableau T, rooted at a node of T such that all OR-nodes in T' have exactly one (AND-node) successor from T in T', and all AND-nodes in T' either have no successors in T', or, have all their (OR-node) successors from T in T'.

If the root node of the tableau is deleted, we halt and declare the specification ϕ as inconsistent (unsatisfiable). If not, we proceed to the next step.

7.2.3 Obtaining a Model $\mathcal{M}$ from T_ϕ

A model $\mathcal{M}$ is obtained by joining together model fragments rooted at AND-nodes of T_ϕ: each model fragment is a rooted DAG of AND-nodes embeddable in T_ϕ such that all eventuality formulas labeling the root node are fulfilled in the fragment. We do not explain this step in more detail, as it is identical to the procedure in [47] [3]. After extracting $\mathcal{M}$ from T_ϕ, we modify the labels of the states of $\mathcal{M}$ by eliminating all labels other than simple atoms, identifying the values of the program variables in each state of $\mathcal{M}$. If there exist n states $s_1, \ldots, s_n$ with the exact same labels after this step, we introduce an auxiliary variable x with domain $\{0, 1, 2, \ldots, n\}$ to distinguish between the states: x is assumed to be 0 in all states other than $s_1, \ldots, s_n$; for each $j \in \{1, \ldots, n\}$, we set x to j in transitions into s_j, and set x back to 0 in transitions out of s_j. This completes the model generation. $\mathcal{M}$ is guaranteed to satisfy ϕ by construction.

[3] There may be multiple models embedded in T_ϕ. In [47], in order to construct model fragments, whenever there are multiple sub-DAGs rooted at an OR-node B that fulfill the eventualities labeling B, one of minimal size is chosen, where size of a sub-DAG is defined as the length of its longest path. There are other valid criteria for choosing models, exploring which is beyond the scope of this work.

7.2.4 Decomposition of $\mathcal{M}$ into $\mathcal{P}_1^s$ and $\mathcal{P}_2^s$

Recall that $\mathcal{P}_1$ and $\mathcal{P}_2$ are unsynchronized processes with atomic statements such as assignments, condition tests and gotos, and no CCRs. In this last step of our basic algorithmic framework, we generate $\mathcal{P}_1^s$ and $\mathcal{P}_2^s$ consisting of CCRs, enclosing each atomic statement of $\mathcal{P}_1$ and $\mathcal{P}_2$.

Without loss of generality, consider location λ_1 in $\mathcal{P}_1$. The guard for the CCR for $stmt(\lambda_1)$ in $\mathcal{P}_1^s$ corresponds to all states in $\mathcal{M}$ in which $stmt(\lambda_1)$ is *enabled*, i.e., states in which $\mathcal{P}_1$ is at location λ_1 and from which there exists a $\mathcal{P}_1$ transition. To be precise, $stmt(\lambda_1)$ is enabled in state s in $\mathcal{M}$ iff there exists a transition $(s, s') \in R$ such that $val^s[loc_1] = \lambda_1$, $val^{s'}[loc_1] = \lambda_1'$ with λ_1' being a valid next location for $\mathcal{P}_1$, and, $val^s[loc_2] = val^{s'}[loc_2]$. The guard G_s corresponding to such a state s is the valuation of all program variables other than loc_1 in state s. Thus, if $val^s[loc_2] = \ell_2$ and for all $v_j \in Var = \{v_1, \ldots, v_h\}$, $val^s[v_j] = \mu_j$, then G_s is given by $(loc_2 = \lambda_2) \wedge \bigwedge_{j=1}^{h} v_j = \mu_j$.

If $\mathcal{M}$ does not contain an auxiliary variable, then the CCR for $stmt(\lambda_1)$ in $\mathcal{P}_1^s$ is simply $G_{1,1}? \rightarrow stmt(\lambda_1)$, where $G_{1,1}$ is the disjunction of guards G_s corresponding to all states s in $\mathcal{M}$ in which $stmt(\lambda_1)$ is enabled. However, if $\mathcal{M}$ contains an auxiliary variable x (with domain $\{0, 1, 2, \ldots, n\}$), then one may also need to perform updates to x within the CCR instruction block. In particular, if $stmt(\lambda_1)$ is enabled on state s in $\mathcal{M}$, transition (s, s') in $\mathcal{M}$ is a $\mathcal{P}_1$ transition, and if there is an assignment $x := j$ for some $j \in \{0, \ldots, n\}$ along transition (s, s'), then besides $stmt(\lambda_1)$, the statement block of the CCR

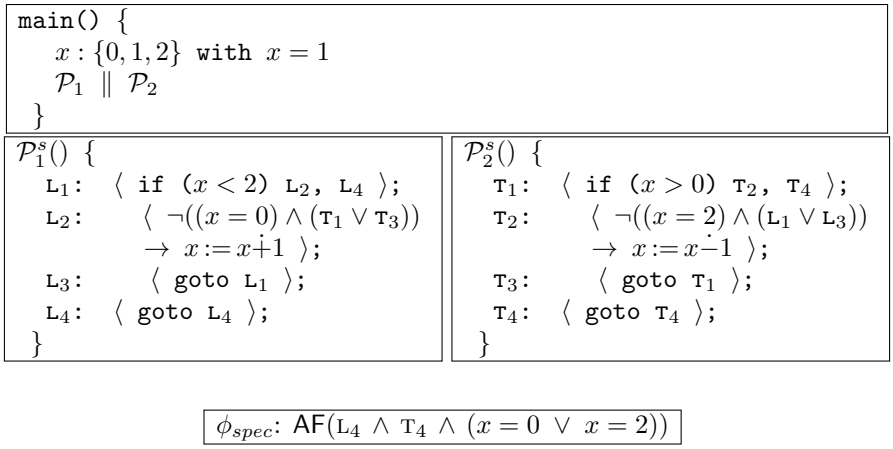


Figure 7.3: Synchronized concurrent program $\mathcal{P}^s$ such that $\mathcal{P}^s \models \phi_{spec}$

for $stmt(\lambda_1)$ in P_1^s includes instructions in our programming language corresponding to: `if` $(G_s)\ x := j$.

The synchronized process $\mathcal{P}_1^s$ (and similarly $\mathcal{P}_2^s$) can be generated by inserting a similarly generated CCR at each location in $\mathcal{P}_1$ (and $\mathcal{P}_2$). The modified concurrent program $\mathcal{P}_s$ is given by $\mathcal{P}_s :: [\texttt{declaration}]\ [\mathcal{P}_1^s \parallel \mathcal{P}_2^s]$, where the declaration includes auxiliary variable x with domain $\{0, 1, 2, \ldots, n\}$ if $\mathcal{M}$ contains x with domain $\{0, 1, 2, \ldots, n\}$.

Example: For the example concurrent program and specification from Fig. 7.2, we obtain the synchronized concurrent program shown in Fig. 7.3. Observe the CCRs introduced in locations L_2 and T_2, respectively.

7.2.5 Algorithm Notes

The following theorems assert the correctness of our basic algorithmic framework for synthesizing synchronization for unsynchronized processes $\mathcal{P}_1$, $\mathcal{P}_2$, as defined in Sec. 7.1.2, with the restriction that all program variables are shared variables that are initialized to specific values. These theorems follow from the correctness of the synthesis of synchronization skeletons [46, 47], and the modifications made in this chapter to accommodate **L**CTL specifications over the program variables.

Theorem 7.2.1. *Given unsynchronized processes $\mathcal{P}_1$, $\mathcal{P}_2$ and an* **L**CTL *formula ϕ_{spec}, if our basic algorithm generates $\mathcal{P}^s$, then $\mathcal{P}^s \models \phi_{spec}$.*

Theorem 7.2.2. *Given unsynchronized processes $\mathcal{P}_1$, $\mathcal{P}_2$, and an* **L**CTL *formula ϕ_{spec}, if the temporal specification $\phi = \phi_{spec} \wedge \phi_P$ is consistent as a whole, then our method constructs $\mathcal{P}^s$ such that $\mathcal{P}^s \models \phi_{spec}$.*

The complexity of our method is exponential in the size of ϕ, i.e., exponential in the size of ϕ_{spec} and the number of program variables V.

7.3 Extensions

In this section, we demonstrate the adaptability of our basic algorithmic framework by considering more general program models. In particular, we discuss extensions for synthesizing correct synchronization in the presence of uninitialized variables and local variables. Furthermore, we extend our framework

to programming languages with locks and `wait/notify` operations over condition variables by presenting an automatic compilation of CCRs into synchronization code based on these lower-level synchronization primitives (similar to Chapter 6). We conclude with an extension of the framework to multiple processes.

7.3.1 Uninitialized Variables

In Sec. 7.2, we assumed that all data variables are initialized to specific values over their domains. This assumption may not be satisfied in general as it disallows any kind of user or environment input to a concurrent program. In the program model presented in Sec. 7.1.2, only some (or even none) of the data variables may be initialized to specific values within the program. This is a more realistic setting, which allows a user or environment to choose the initial values of the remaining data variables. In this subsection, we present a simple, brute-force extension of our basic algorithm for synthesizing synchronization in the presence of uninitialized variables.

The formula $\phi_{\mathcal{P}}$, expressing the concurrency and operational semantics of $\mathcal{P}$, remains the same, except for the initial condition. Instead of a single initial state, the initial condition in $\phi_{\mathcal{P}}$ specifies the set of all possible initial states, with the control and initialized data variables set to their initial values, and the remaining data variables ranging over all possible values in their respective domains. Let us denote by Var_{inp} this remaining set of data variables, that are, essentially, inputs to the program $\mathcal{P}$. The set of program-initialized

data variables is then $Var \setminus Var_{inp}$. The initial condition in ϕ_P is expressed as:

$$\bigwedge_k val[loc_k] = \lambda_k^0 \ \wedge \bigwedge_{v \in Var \setminus Var_{inp}} (v = v_{init}) \ \wedge \bigwedge_{v \in Var_{inp}} \bigvee_{\mu \in D_v} (v = \mu),$$

where D_v is the domain of v.

The root node of the tableau T_ϕ is now an AND-node with multiple OR-node successors, each corresponding to a particular valuation $\boldsymbol{\mu}$ of all the data variables (the values of the control variable and initialized data variables are the same in any such valuation). Each such OR-node yields a model $M_{\boldsymbol{\mu}}$ for the formula ϕ, and a corresponding decomposition of $M_{\boldsymbol{\mu}}$ into synchronized processes $\mathcal{P}^s_{1_{\boldsymbol{\mu}}}$ and $\mathcal{P}^s_{2_{\boldsymbol{\mu}}}$.

To generate synchronized processes $\mathcal{P}^s_1$ and $\mathcal{P}^s_2$ such that for all possible initial valuations $\boldsymbol{\mu}$ of the data variables, $\mathcal{P}^s \models \phi_{spec}$, we propose to *unify* the CCRs corresponding to each valuation $\boldsymbol{\mu}$ as follows:

1. Introduce a new variable v_{copy} for every input data variable v in Var_{inp}. Declare v_{copy} as a variable with the same domain as v. Assign v_{copy} the (input) value of v.

2. Replace every CCR guard G in the synchronized process $\mathcal{P}^s_{k_{\boldsymbol{\mu}}}$ with the guard $G_{\boldsymbol{\mu}}$, given by, $\bigwedge_{v \in Var_{inp}} (v_{\mathsf{copy}} = \mu_v) \wedge G$, where the valuation of v in $\boldsymbol{\mu}$ is $\boldsymbol{\mu}_v$. Similarly, update every conditional guard accompanying an auxiliary variable assignment within a CCR instruction block in $\mathcal{P}^s_{k_{\boldsymbol{\mu}}}$.

3. The unified guard for each CCR in $\mathcal{P}_1^s$ and $\mathcal{P}_2^s$ is given by the disjunction of the corresponding guards $G_{\boldsymbol{\mu}}$ in all $\mathcal{P}_{1_{\boldsymbol{\mu}}}^s$ and $\mathcal{P}_{2_{\boldsymbol{\mu}}}^s$. The unified conditional guards for auxiliary variable updates in the CCR instruction blocks are computed similarly.

Note that the unified guards inferred above, as well as in Sec. 7.2.4, may not in general be *readable* or *compact*. However, since each guard is expected to be an **L**-term over a finite set of variable, function and predicate symbols with known interpretations, it is possible to obtain a simplified **L**-term with the same value as the guard. This translation is beyond the scope of this work, but we refer the reader to [83] for a similar approach.

7.3.2 Local Variables

Another assumption in Sec. 7.2 was that all program variables, including control variables, were shared variables. Since one typically associates a cost with each shared variable access, it is impractical to expect all program variables to be shared variables. This is especially true of control variables, which are generally never declared explicitly or accessed in programs. Thus, the guards inferred in Sec. 7.2.4, ranging over locations of the other process, are somewhat irregular. Indeed, any guard for a process $\mathcal{P}_k$ must only be defined over the data variables Var_k accessible by $\mathcal{P}_k$. In what follows, we discuss various solutions to address this issue.

Let us assume that we have a model $\mathcal{M} = (S, R, L)$ for ϕ, with states labeled by the valuations of the control variables Loc, the shared data variables

X, the local data variables $Y = \bigcup_k Y_k$, and possibly a shared auxiliary variable x. For the purpose of this subsection, let x be included in the set X. We first check if the set of states S of $\mathcal{M}$ has the property that for any two states $s_1,\ s_2$ in S: $\left(\bigwedge_{loc \in Loc} val^{s_1}[loc] = val^{s_2}[loc] \ \wedge \ \bigwedge_{y \in Y} val^{s_1}[y] = val^{s_2}[y]\right) \ \Leftrightarrow \ \bigwedge_{x \in X} val^{s_1}[x] = val^{s_2}[x]$. If this is true, then each state $s \in S$ is uniquely identified by its valuation of the shared data variables X. We can then simply factor out guards from $\mathcal{M}$ for each process that only range over X, without missing out on any permitted behaviour in $\mathcal{M}$. If this is not true, we can perform other similar checks. For instance, we can check if for a particular k: any two states in S match in their valuations of the variables $\{loc_k\} \cup Y_k \cup X$ iff they match in their valuations of the other program variables. If this is true, then the process $\mathcal{P}_k$ can distinguish between states in S by the valuations of its variables $Var_k \cup \{loc_k\}$. Thus, we can infer guards for $\mathcal{P}_k$, that are equivalent to the guards inferred in Sec. 7.2.4, but only range over Var_k.

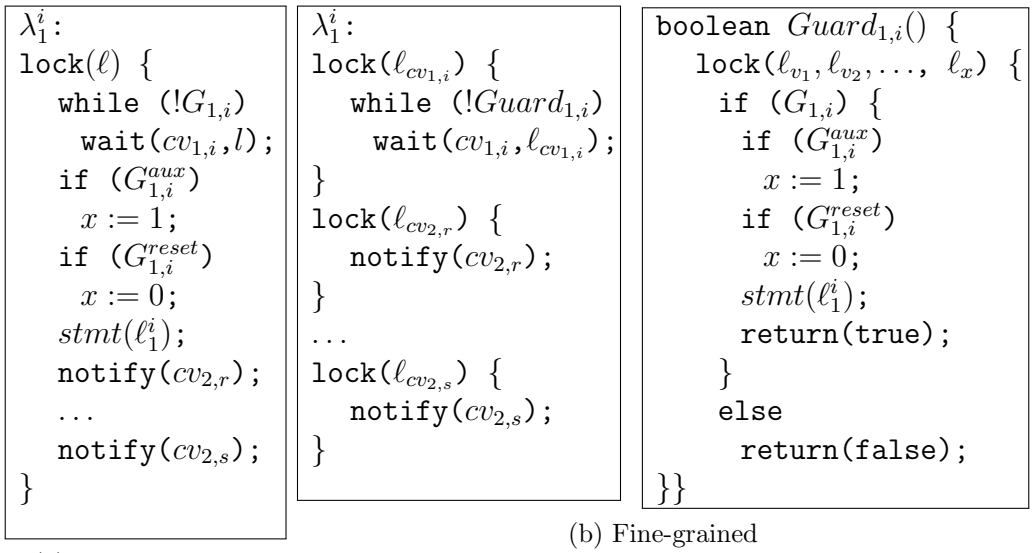
In general, however, there will be states $s_1,\ s_2$ in S which cannot be distinguished by the valuations of a particular process's, or worse, by any process's variables. This general situation presents us with a trade-off between synchronization cost and concurrency: we can introduce additional shared variables to distinguish between such states, thereby increasing the synchronization cost and allowing more behaviours of $\mathcal{M}$ to be preserved in $\mathcal{P}^s$, or, we can resign to *limited observability* [126] of global states, resulting in lower synchronization cost and fewer permitted behaviours of $\mathcal{M}$. In particular, for the latter case, we implement a safe subset of the behaviours of $\mathcal{M}$ by inferring

synchronization guards corresponding to the negation of variable valuations (states) that are not present in $\mathcal{M}$. Since a global state $u \notin \mathcal{M}$ may be indistinguishable over some Var_k from a state $s \in \mathcal{M}$, when eliminating behaviours rooted at u, we also eliminate all (good) behaviours of $\mathcal{M}$, rooted at s. We refer the reader to [126] for a detailed treatment of this trade-off.

7.3.3 Synchronization using Locks and Condition Variables

While CCRs provide an elegant high-level synchronization solution, many programming languages prefer and only provide lower-level synchronization primitives such as locks for mutual exclusion, and `wait`/`notify` over condition variables for condition synchronization. Similar to Chapter 6, we present an automatic compilation of the CCRs inferred in Sec. 7.2.4 for $\mathcal{P}_1^s$, $\mathcal{P}_2^s$ into both coarse-grained and fine-grained synchronization code based on these lower-level primitives. The resulting processes are denoted as $\mathcal{P}_1^c$, $\mathcal{P}_2^c$ (coarse-grained) and $\mathcal{P}_1^f$, $\mathcal{P}_2^f$ (fine-grained).

In both cases, we declare locks and conditions variables for synchronization. For the program $\mathcal{P}^c$, which has a coarser level of lock granularity, we declare a single lock ℓ for controlling access to shared variables and condition variables. For the program $\mathcal{P}_1^f \parallel \mathcal{P}_2^f$ with a finer level of lock granularity, we declare separate locks ℓ_v, ℓ_x for controlling access to each shared data variable $v \in X$ and the shared auxiliary variable x, respectively. We further define a separate lock $\ell_{cv_{1,i}}$, $\ell_{cv_{2,j}}$ for each condition variable $cv_{1,i}$, $cv_{2,j}$ to allow simultaneous processing of different condition variables.



(a) Coarse-grained

(b) Fine-grained

Figure 7.4: Coarse and fine-grained synchronization code corresponding to an example CCR at location ℓ_1^i of $\mathcal{P}_1$. Guards $G_{1,i}^{aux}$, $G_{1,i}^{reset}$ correspond to all states in $\mathcal{M}$ on which $stmt(\ell_1^i)$ is enabled, and there's an assignment x:=1, x:=0, respectively, along a $\mathcal{P}_1$ transition out of the states.

We refer the reader to Fig. 7.4 for an example of coarse-grained and fine-grained synchronization code corresponding to the CCR at location λ_1^i of $\mathcal{P}_1$. Note that, for ease of presentation, we have used conventional pseudocode, instead of our programming language. Since these compilations are very similar to the ones presented in Chapter 6, we do not describe them further.

7.3.4 Multiple ($K > 2$) Processes

Our basic algorithmic framework can be extended, similar to the guidelines presented in Sec. 6.3.3, for synthesizing synchronization for concurrent pro-

grams with an arbitrary (but fixed) number K of processes. While the naive extension may exhibit a state explosion problem, we can adapt the more scalable synthesis algorithms presented in [3,4] to the synthesis of $\mathbf{L}CTL$ formulas. Also, note that the compilation of CCRs into coarse-grained and fine-grained synchronization code acts on individual processes directly, without construction or manipulation of the global model, and hence circumvents the state-explosion problem for arbitrary K.

Chapter 8

Bibliographic Notes

Automatic synthesis of programs has received a fair share of attention from the formal methods community over the last three decades. While researchers have proposed synthesis algorithms for both sequential (see [59] for a survey) and concurrent programs, in what follows, we mainly restrict our attention to relevant algorithms proposed for synthesis of synchronization for concurrent systems.

Synthesis from temporal logic specifications. Early work in this domain included work on synthesis for CTL specifications [47] and LTL specifications [91], using tableau-based decision procedures. While the core algorithm presented in [47] was promising, a primary limitation of the framework was its remoteness from realistic concurrent programs and programming languages. The limited modeling of shared-memory concurrency in this work did not include local and shared data variables, and hence, could not support specifications over the values of program variables. There was no explicit treatment of processes with branching, observability of program counters or local variables, and no attempt to synthesize synchronization based on lower-level synchro-

nization primitives available in modern programming languages. In contrast, our approaches

More recently, practically viable synthesis of synchronization has been proposed for both finite-state [126] and infinite-state concurrent programs [127]. However, in both [126], [127], the authors only handle safety specifications; in fact, it can be shown that synthesis methods that rely on pruning a global product graph [72, 126, 127] cannot, in general, work for liveness. Moreover, these papers do not support any kind of external environment; in particular, these papers do not account for different (environment-enabled) initializations of the program variables. Finally, similar to [47], these papers only synthesize high-level synchronization in the form in CCRs [126] and atomic sections [127], and do not attempt to synthesize synchronization based on lower-level synchronization primitives available in commonly used programming languages.

On the other end of the spectrum, there has been some important work on automatic synthesis of lower-level synchronization, in the form of memory fences, for concurrent programs running on relaxed memory models [84, 86]. There has also been work on mapping high-level synchronization into lower-level synchronization [32,132]. These paper do not prove that their high-to-low compilations preserve correctness, and do not, in general, pursue correctness by design. Instead they rely on *verification-driven* frameworks, which involve verifying either the synthesized implementation [32] or the manually-written high-level implementation [132]. These papers do not treat liveness proper-

ties and do not address refinement of locking granularity for access to shared variables.

In contrast to the above approaches, the synchronization synthesis framework presented in Chapter 7 (also presented in [106]) (a) supports both safety and liveness properties over control and data variables, (b) supports finite-state concurrent programs composed of processes that may have local and shared variables, may be straight-line or branching programs, may be ongoing or terminating, and may have program-initialized or user-initialized variables, (c) is fully algorithmic, and (d) generates synchronized concurrent programs that are correct-by-construction. The framework presented in Chapter 6 (also presented in [48]) (a) caters for both safety and liveness, (b) is fully algorithmic, (c) automatically construct a high-level synchronization solution, (d) can generate a low level solution based on widely used synchronization primitives, (e) can generate both coarse-grained and fine-grained low-level solutions, and (f) is provably sound and complete.

Synthesis of fine-grained synchronization. Among papers that address refinement of locking granularity are [5] in which the authors provide a methodology to refine the granularity of atomicity in the synchronization skeletons of [47] to generate synchronization in which each instruction is an atomic read or an atomic write. More recently, a multitude of papers ([25, 49] etc.) have addressed the problem of compiler-based lock inference for atomic sections. While these papers propose different levels of automation and lock granular-

ity, they do not, in general, support condition variables, and often rely on the availability of high-level synchronization in the form of atomic sections. Nevertheless, it may be possible and useful to adapt some of the ideas (multi-granular locks, optimization with respect to costs) introduced in these papers to our framework.

Sketching. Sketching [114, 115] is a search-based program synthesis technique that can also be used for synthesizing optimized implementations of synchronization primitives, e.g. barriers, from partial program sketches. In contrast to the correctness-by-design paradigm in our work, sketching is also a *verification-driven* approach.

Synthesis based on discrete control theory. The authors in [41] and [129] synthesize maximally-permissive concurrency control for regular language (safety) specifications, and deadlock-avoidance, respectively using discrete control theory. The authors in [7] handle a richer class of specifications expressible as Petri nets for concurrent programs with dynamically instantiated/terminated threads, using a limited lookahead-based supervisory control algorithm. Since the attitude in [7] is one of avoidance of constraint violation rather than strict prevention, in the presence of limited lookahead, their framework may not always be maximally permissive. Note that our framework can be made maximally permissive. Finally, none of these papers address fine-grained concurrency control.

Besides the above topics, we mention that there also has been work on automatic inference of synchronization for specific types of bugs such as

atomicity violations [73, 79], trace-based synthesis of various concurrent program transformations (including lock insertion) for safety properties [124], synthesis of Java monitor classes from process algebraic specifications [17] and component- and interface-based synthesis [8, 14].

A note on reactive systems. A shared-memory concurrent program can also be viewed as a *reactive system*. A *reactive system* [64,101] is described as one that maintains an ongoing interaction with an external environment or within its internal concurrent modules. Such systems cannot be adequately described by relational specifications over initial and final states - this distinguishes them from transformational or relational programs. An adequate description of a reactive system must refer to its ongoing desired behaviour, throughout its (possibly non-terminating) activity - temporal logic [100] has been recognized as convenient for this purpose.

A reactive system may be terminating or not, sequential or concurrent, and implemented on a monolithic or distributed architecture. A reactive system can also be open or closed [102, 103]. This has been a somewhat overlooked dichotomy in recent years. We have observed that it is not uncommon to view reactive systems exclusively as open systems; this is especially true in the context of synthesis. While the first algorithms on synthesis of concurrent programs [4,47,91] were proposed for closed reactive systems, the foundational work in [102, 103] set the stage for an extensive body of impressive results on synthesis of open reactive systems (see [85, 120] for surveys).

We contend that the relatively simpler problem of synthesis of closed

reactive systems is an important problem in its own right. This is especially true in the context of shared-memory concurrent programs, where it is sometimes sufficient and desirable to model programs as closed systems and force the component processes to cooperate with each other for achieving a common goal. If one must model an external environment, it is also often sufficient to model the environment in a restricted manner (as in this paper) or optimistically assume a helpful environment (see [29]).

Part IV

Robustness Analysis

In Part II and Part III, we presented program debugging and synthesis algorithms targeting traditional *qualitative* correctness properties such as safety and liveness. In Part I, we have seen that today's software systems often operate in *uncertain* environments. Left unchecked, such uncertainty can lead to highly unpredictable system behaviour. Thus, a program may have a *correct* execution on every individual input, but its output may be highly sensitive to the minutest perturbation in its operating environment.

In this part of the dissertation, we focus on checking if a system is *robust* — small perturbations to the operating environment of the system do not change the system's observable behavior substantially. Reasoning about system robustness demands a departure from techniques to analyze and develop traditionally correct systems, as the former requires *quantitative* reasoning about the system behavior. We target robustness analysis of two classes of systems — string transducers and networked systems. For each system, we define robustness of the system with respect to a specific source of uncertainty. In particular, we analyze the behaviour of transducers in the presence of input perturbations, and the behaviour of networked systems in the presence of channel perturbations. Our overall approach is automata-theoretic. We present decision procedures based on reducing the problem of robustness verification of our systems to the problem of checking the emptiness of carefully constructed automata. Thus, the system under consideration is robust if and only if the language of a particular automaton is empty.

In Chapter 9, we define the relevant transducer models and distance

metrics, and present constructions for various *distance-tracking* automata. In Chapter 10, we present decision procedures for robustness analysis of string transducers with respect to input perturbations. Changes to input and output strings are quantified using weighted generalizations of the Manhattan and Levenshtein distances over strings. In Chapter 11, we present decision procedures for robustness analysis of networked systems, when the underlying network channels are prone to errors. The distance metrics considered are the Manhattan and Levenshtein distances over strings. Finally, we conclude this part with a discussion of related work in Chapter 12.

Chapter 9

Groundwork

Overview. In this chapter, we begin by defining the transducer models and distance metrics considered in this part of the dissertation. Key components of our approach to robustness analysis of systems are machines that can track distances between two strings; some of these machines are *reversal-bounded counter machines*. Hence, we also review reversal-bounded counter machines, and present constructions for various *distance-tracking* automata.

In what follows, we use the following notation. Input strings are typically denoted by lowercase letters s, t etc. and output strings by s', t' etc. We denote the concatenation of strings s and t by $s.t$, the i^{th} character of string s by $s[i]$, the substring $s[i].s[i+1].\ldots.s[j]$ by $s[i,j]$, the length of the string s by $|s|$, and the empty string and empty symbol by ϵ.

9.1 Functional Transducers

A *transduction* R from a finite alphabet Σ to a finite alphabet Γ is an arbitrary subset of $\Sigma^\star \times \Gamma^\star$. We use $R(s)$ to denote the set $\{t \,|\, (s,t) \in R\}$. We say that a transduction is *functional* if $\forall s \in \Sigma^\star$, $|R(s)| \leq 1$.

A *finite transducer* (FT) is a finite-state device with two tapes: a read-only input tape and a write-only output tape. An FT scans the input tape from left to right, and in each state reads an input symbol, possibly changes state, writes a finite string to the output tape, and advances its reading head one position to the right. In each such step, an FT nondeterministically chooses its next state and an output string to write. The output of an FT is the string on the output tape if the FT finishes scanning the input tape in some designated final state. Formally, a finite transducer $\mathcal{T}$ is a tuple $(Q, \Sigma, \Gamma, q_0, E, F)$ where Q is a finite nonempty set of states, $q_0 \in Q$ is the initial state, E is a set of transitions defined as a finite subset of $(Q \times \Sigma \times \Gamma^\star \times Q)$, and $F \subseteq Q$ is a set of final states[1].

A run of $\mathcal{T}$ on a string $s = s[1]s[2]\ldots s[n]$ is defined in terms of the sequence: (q_0, ϵ), (q_1, w'_1), $\ldots$, (q_n, w'_n), where for each i, $1 \leq i \leq n$, $(q_{i-1}, s[i], w'_i, q_i)$ is a transition in E. A run is called accepting if $q_n \in F$. The output of $\mathcal{T}$ along a run is the string $w'_1.w'_2.\ldots.w'_n$ if the run is accepting, and is undefined otherwise. The transduction computed by an FT $\mathcal{T}$ is the relation $[\![\mathcal{T}]\!] \subseteq \Sigma^\star \times \Gamma^\star$, where $(s, s') \in [\![\mathcal{T}]\!]$ iff there is an accepting run of $\mathcal{T}$ on s with s' as the output along that run. $\mathcal{T}$ is called *single-valued* or *functional* if $[\![\mathcal{T}]\!]$ is functional. Thus, a functional transducer may be nondeterministic, but must define a function between regular sets of strings. Checking if an arbitrary FT

[1]Some authors prefer to call this model a *generalized sequential machine*. In contrast, a transducer is defined to allow ϵ-transitions; thus, a transducer is allowed to change state without moving the reading head. Note that ϵ-transitions are disallowed in our definition of transducers.

is functional can be done in polynomial time [60]. The input language, $\mathcal{L}$, of a functional transducer $\mathcal{T}$ is the set $\{s \mid [\![\mathcal{T}]\!](s) \text{ is defined}\}$. When viewed as a relation over $\Sigma^\star \times \Gamma^\star$, $[\![\mathcal{T}]\!]$ defines a partial function; however, when viewed as a relation over $\mathcal{L} \times \Gamma^\star$, $[\![\mathcal{T}]\!]$ is a total function.

Mealy Machines. These are deterministic, symbol-to-symbol, functional transducers. Thus, from every state $q \in Q$, there exists exactly one transition, and every transition in E is of the form (q, a, w', q') with $|w'| = 1$. The input language $\mathcal{L}$ of a Mealy machine $\mathcal{T}$ is the set $\Sigma^\star$ (i.e., every state is accepting). Hence, the transduction implemented by $\mathcal{T}$ is a total function $[\![\mathcal{T}]\!] : \Sigma^\star \to \Gamma^\star$.

In what follows, we use the term finite transducers, or simply transducers, to refer to both functional transducers and Mealy machines, and distinguish between them as necessary. As a technicality that simplifies our proofs, we assume that for all $i > |s|$, $s[i] = \texttt{\#}$, where # is a special end-of-string symbol not in Σ or Γ.

9.2 Distance Metrics

A *metric space* is an ordered pair (M, d), where M is a set and $d : M \times M \to \mathbb{R}$, the distance metric, is a function with the properties: (1) $d(x, y) \geq 0$, (2) $d(x, y) = 0$ iff $x = y$, (3) $\forall x, y : d(x, y) = d(y, x)$, and (4) $\forall x, y, z : d(x, z) \leq d(x, y) + d(y, z)$.

The Hamming distance and Levenshtein distance metrics are popular distance metrics for measuring distances (or similarity) between strings. The

Hamming distance, defined for two equal length strings, is the minimum number of symbol substitutions required to transform one string into the other. For strings of unequal length, the Hamming distance is replaced by its natural extension — the Manhattan distance or L_1-norm — which also accounts for the difference in the lengths. In particular, the Manhattan distance $d_M(s,t)$ between strings s and t measures the number of positions in which s and t differ, and can be defined using the following recurrence relations, for $i \geq 1$, and $s[0] = t[0] = \epsilon$:

$$\begin{array}{l} d_M(s[0], t[0]) = 0 \\ d_M(s[0,i], t[0,i]) = d_M(s[0,i\text{-}1], t[0,i\text{-}1]) + \texttt{diff}(s[i], t[i]) \end{array} \tag{9.1}$$

where, $\texttt{diff}(a,b)$ is defined to be 0 is $a = b$ and 1 otherwise.

The Levenshtein distance $d_L(s,t)$ between strings s and t is the minimum number of symbol insertions, deletions and substitutions required to transform one string into another. The Levenshtein distance, also called the edit distance, is defined by the following recurrence relations, for $i, j \geq 1$, and $s[0] = t[0] = \epsilon$:

$$\begin{array}{lll} d_L(s[0], t[0]) = 0, & d_L(s[0,i], t[0]) = i, & d_L(s[0], t[0,j]) = j \\ d_L(s[0,i],\ t[0,j]) = \min(& d_L\,(s[0,i\text{-}1],\ t[0,j\text{-}1]) + \texttt{diff}(s[i],\ t[j]), & \\ & d_L\,(s[0,i\text{-}1],\ t[0,j]) + 1, & \\ & d_L\,(s[0,i],\ t[0,j\text{-}1]) + 1) & \end{array} \tag{9.2}$$

The first three relations, that involve empty strings, are obvious. The Levenshtein distance between the nonempty prefixes, $s[0,i]$ and $t[0,j]$, is the minimum over the distances corresponding to three possible transformations: (1) optimal (Levenshtein) transformation of $s[0,i\text{-}1]$ into $t[0,j\text{-}1]$ and substitution

of $s[i]$ with $t[j]$ for an additional cost of $\texttt{diff}(a, b)$, (2) optimal transformation of $s[0, i\text{-}1]$ into $t[0, j]$ and deletion of $s[i]$ for an additional cost of 1 (3) optimal transformation of $s[0, i]$ into $t[0, j\text{-}1]$ and insertion of $t[j]$ for an additional cost of 1.

Example. Consider the strings $s = baa$ and $t = abca$. We have $d_M(s, t) = 4$, $d_L(s, t) = 2$. The Manhattan transformation of s into t simply involves 4 substitutions. The Levenshtein distance of 2 is obtained by aligning the strings as: $\begin{matrix} \sqcup & b & a & a \\ a & b & c & a \end{matrix}$ where $\sqcup$ is a special place-holder symbol. Thus, a Levenshtein transformation of s into t involves inserting the symbol a before s and substituting the symbol a in s with the symbol c.

The Hamming/Manhattan and Levenshtein distances only track the number of symbol mismatches, and not the degree of mismatch. For some applications, these distance metrics can be too coarse. Hence, we also consider distance metrics equipped with integer penalties - *pairwise symbol mismatch penalties* for substitutions and a *gap penalty* for insertions/deletions. We denote by $\texttt{gdiff}(a, b)$ the mismatch penalty for substituting symbols a and b, with $\texttt{gdiff}(a, b) = 0$ if $a = b$. We require $\texttt{gdiff}(a, b)$ to be well-defined when either a or b is `#`. We denote by α the fixed, non-zero gap penalty for insertion or deletion of a symbol. We now define these weighted extensions of the Manhattan and Levenshtein distances formally.

The *generalized* Manhattan distance $d_{gM}(s, t)$ between strings s and t is defined by the following recurrence relations, for $i \geq 1$, and $s[0] = t[0] = \epsilon$:

$$
\begin{aligned}
&d_{gM}(s[0], t[0]) = 0 \\
&d_{gM}(s[0,i], t[0,i]) = d_{gM}(s[0,i\text{-}1], t[0,i\text{-}1]) + \texttt{gdiff}(s[j], t[i]).
\end{aligned}
\tag{9.3}
$$

The generalized Levenshtein distance $d_{gL}(s,t)$ between strings s and t is defined by the following recurrence relations, for $i, j \geq 1$, and $s[0] = t[0] = \epsilon$:

$$
\begin{aligned}
&d_{gL}(s[0], t[0]) = 0, \qquad d_{gL}(s[0,i], t[0]) = i\alpha, \qquad d_{gL}(s[0], t[0,j]) = j\alpha \\
&d_{gL}(s[0,i],\ t[0,j]) = \min(\quad d_{gL}\,(s[0,i\text{-}1],\ t[0,j\text{-}1]) + \texttt{gdiff}(s[i],\ t[j]), \\
&\qquad\qquad\qquad\qquad\quad d_{gL}\,(s[0,i\text{-}1],\ t[0,j]) + \alpha, \\
&\qquad\qquad\qquad\qquad\quad d_{gL}\,(s[0,i],\ t[0,j\text{-}1]) + \alpha).
\end{aligned}
\tag{9.4}
$$

The generalized Levenshtein distance between the nonempty prefixes, $s[0,i]$ and $t[0,j]$, is the minimum over the distances corresponding to three possible transformations: (1) optimal (generalized Levenshtein) transformation of $s[0,i\text{-}1]$ into $t[0,j\text{-}1]$ and substitution of $s[i]$ with $t[j]$, with a mismatch penalty of $\texttt{gdiff}(a,b)$, (2) optimal transformation of $s[0,i\text{-}1]$ into $t[0,j]$ and deletion of $s[i]$, with a gap penalty of α, and, (3) optimal transformation of $s[0,i]$ into $t[0,j\text{-}1]$ and insertion of $t[j]$ with a gap penalty of α.

Example. Consider the strings $s = baa$ and $t = abca$ again. Let: $\texttt{gdiff}(a,b) = \texttt{gdiff}(b,c) = 1$, $\texttt{gdiff}(a,c) = 2$ and $\alpha = 1$. We have $d_{gM}(s,t) = 5$ and $d_{gL}(s,t) = 3$, using the same transformations as in the previous example.

Observe that if $\texttt{gdiff}(a,b)$ is defined to be equal to $\texttt{diff}(a,b)$, the definitions for the generalized Manhattan and Levenshtein distances correspond to the definitions for the usual Manhattan and Levenshtein distances in equations (9.1) and (9.2), respectively. In our work, we assume that $\texttt{gdiff}(a,b)$

and α are external parameters provided to the algorithm by the user, and we require that the resulting generalized Manhattan and Levenshtein distances are distance metrics.

9.3 Reversal-bounded Counter Machines

A (one-way, nondeterministic) h-counter machine [68, 69] $\mathcal{A}$ is a (one-way, nondeterministic) finite automaton, augmented with h integer counters. Let G be a finite set of integer constants (including 0). In each step, $\mathcal{A}$ may read an input symbol, perform a test on the counter values, change state, and increment each counter by some constant $g \in G$. A test on a set of integer counters $Z = \{z_1, \ldots, z_h\}$ is a Boolean combination of tests of the form $z\theta g$, where $z \in Z$, $\theta \in \{\leq, \geq, =, <, >\}$ and $g \in G$. Let $\mathcal{T}_Z$ be the set of all such tests on counters in Z.

Formally, $\mathcal{A}$ is defined as a tuple $(\Sigma, X, x_0, Z, G, E, F)$ where Σ, X, x_o, F, are the input alphabet, set of states, initial state, and final states respectively. Z is a set of h integer counters, and $E \subseteq X \times (\Sigma \cup \epsilon) \times \mathcal{T}_Z \times X \times G^h$ is the transition relation. Each transition $(x, \sigma, t, x', g_1, \ldots, g_h)$ denotes a change of state from x to x' on symbol $\sigma \in \Sigma \cup \epsilon$, with $t \in \mathcal{T}_Z$ being the enabling test on the counter values, and $g_k \in G$ being the amount by which the k^{th} counter is incremented.

A configuration of a one-way multi-counter machine is defined as the tuple $(x, \sigma, z_1, \ldots, z_h)$, where x is the state of the automaton, σ is a symbol of the input string being read by the automaton and $z_1, \ldots, z_h$ are the

values of the counters. We define a move relation $\rightarrow_{\mathcal{A}}$ on the configurations: $(x, \sigma, z_1, \ldots, z_h) \rightarrow_{\mathcal{A}} (x', \sigma', z'_1, \ldots, z'_h)$ iff $(x, \sigma, t(z_1, \ldots, z_h), x', g_1, \ldots, g_h) \in E$, where, $t(z_1, \ldots, z_h)$ is *true*, $\forall k$: $z'_k = z_k + g_k$, and σ' is the next symbol in the input string being read. A path is an element of $\rightarrow^{\star}_{\mathcal{A}}$, i.e., a path is a finite sequence of configurations $\mu_1, \mu_2, \ldots$ where for all i : $\mu_i \rightarrow_{\mathcal{A}} \mu_{i+1}$. A run of $\mathcal{A}$ on a string $s = s[1]s[2]\ldots s[n]$ can be defined as a path beginning with an initial configuration in which $x = x_0$, $\sigma = s[1]$ and all counters are set to 0. An accepting configuration is one in which $x \in F$ (with the counter values being unconstrained). A string $s \in \Sigma^{\star}$ is accepted by $\mathcal{A}$ if $(x_0, s[1], 0, \ldots 0) \rightarrow^{\star}_{\mathcal{A}}$ $(x, s[j], z_1, \ldots, z_h)$ for some $x \in F$ and $j \leq n$. The set of strings (language) accepted by $\mathcal{A}$ is denoted $\mathcal{L}(\mathcal{A})$.

In general, multi-counter machines can simulate actions of Turing machines (even with just 2 counters). In [68], the author presents a class of counter machines - *reversal-bounded* counter machines - with efficiently decidable properties. A counter is said to be in an increasing mode between two successive configurations if the counter value is the same or increasing, and in a decreasing mode if the counter value is strictly decreasing. We say that a counter is *r-reversal bounded* if the maximum number of times it changes mode (from increasing to decreasing and vice versa) along *any* path is r. We say that a multi-counter machine $\mathcal{A}$ is r-reversal bounded if each of its counters is at most r-reversal bounded. We denote the class of h-counter, r-reversal-bounded machines by NCM(h, r).

Lemma 9.3.1. *[61] The nonemptiness problem for $\mathcal{A}$ in class NCM(h, r) can*

be solved in NLOGSPACE *in the size of* $\mathcal{A}$.

Recall that for all $i > |s|$, $s[i] = \#$. In what follows, let $\Sigma^{\#} = \Sigma \cup \{\#\}$.

9.4 Manhattan Distance-Tracking Automata

We now define automata $\mathcal{D}_M^{=\delta}$, $\mathcal{D}_M^{>\delta}$ that accept pairs of strings (s, t) such that $d_M(s, t) = \delta$, $d_M(s, t) > \delta$, respectively, where $d_M(s, t)$ is the Manhattan distance between s and t. The automata $\mathcal{D}_M^{=\delta}$, $\mathcal{D}_M^{>\delta}$ are 1-reversal-bounded 1-counter machines (i.e., in NCM(1,1)), and are each defined as a tuple $(\Sigma^{\#} \times \Sigma^{\#}, X, x_0, Z, G, E, F)$, where $(\Sigma^{\#} \times \Sigma^{\#})$ is the input alphabet, $X = \{x_0, x, acc\}$, is a set of three states, x_0 is the initial state, $Z = \{z\}$ is a single 1-reversal-bounded counter, $G = \{\delta, 0, -1\}$ is a set of integers, and $F = \{acc\}$ is the singleton set of final states. The transition relations of $\mathcal{D}_M^{=\delta}$, $\mathcal{D}_M^{>\delta}$ both include the following transitions:

1. An *initialization transition* $(x_0, (\epsilon, \epsilon), \mathit{true}, x, \delta)$ that sets the counter z to δ.

2. Transitions of the form $(x, (a, a), z \geq 0, x, 0)$, for $a \neq \#$, that read a pair of identical, non-# symbols, and leave the state and counter of $\mathcal{D}_M^{=\delta}$, $\mathcal{D}_M^{>\delta}$ unchanged.

3. Transitions of the form $(x, (a, b), z \geq 0, x, -1)$, for $a \neq b$, which read a pair (a, b) of distinct symbols, and decrement the counter z by 1.

4. Transitions of the form $(acc, (*, *), *, acc, 0)$, which ensure that the machine stays in its final state upon reaching it.

The only difference in the transition relations of $\mathcal{D}_M^{=\delta}$, $\mathcal{D}_M^{>\delta}$ is in their transitions into accepting states. The *accepting transitions* of $\mathcal{D}_M^{=\delta}$ are of the form $(x, (\#, \#), z = 0, acc, 0)$, and move $\mathcal{D}_M^{=\delta}$ to an accepting state upon reading a $(\#, \#)$ pair when the counter value is zero, i.e., when the Manhattan distance between the strings being read is exactly equal to δ. The accepting transitions of $\mathcal{D}_M^{>\delta}$ are of the form $(x, (*, *), z < 0, acc, 0)$, and move $\mathcal{D}_M^{>\delta}$ to an accepting state whenever the counter value goes below zero, i.e., when the Manhattan distance between the strings being read is greater than δ.

Note that the size of $\mathcal{D}_M^{=\delta}$ or $\mathcal{D}_M^{>\delta}$ is $\mathcal{O}(\delta + |\Sigma|^2)$. The following lemma states the correctness of the above automata constructions.

Lemma 9.4.1. *$\mathcal{D}_M^{=\delta}$, $\mathcal{D}_M^{>\delta}$ accept a pair of strings (s, t) iff $d_M(s, t) = \delta$, $d_M(s, t) > \delta$, respectively.*

Generalized Manhattan Distance-Tracking Automata. The definitions for the automata $\mathcal{D}_{gM}^{=\delta}$, $\mathcal{D}_{gM}^{>\delta}$ that accept pairs of strings (s, t) such that $d_{gM}(s, t) = \delta$, $d_{gM}(s, t) > \delta$, respectively, with $d_{gM}(s, t)$ being the generalized Manhattan distance between s and t, are very similar to the definitions for $\mathcal{D}_M^{=\delta}$, $\mathcal{D}_M^{>\delta}$, respectively. In particular, both $\mathcal{D}_{gM}^{=\delta}$, $\mathcal{D}_{gM}^{>\delta}$ are in NCM(1,1), and are each defined as a tuple $(\Sigma^{\#} \times \Sigma^{\#}, X, x_0, Z, G, E, F)$, where $(\Sigma^{\#} \times \Sigma^{\#})$ is the input alphabet, $X = \{x_0, x, acc\}$, is a set of three states, x_0 is the initial state, $Z = \{z\}$

is a single 1-reversal-bounded counter, $G = \{\delta, 0\} \cup \cup_{a,b\in\Sigma^\#} \{\texttt{gdiff}(a,b)\}$ is a set of integers, and $F = \{acc\}$ is the singleton set of final states. The transition relations of $\mathcal{D}_{gM}^{=\delta}$, $\mathcal{D}_{gM}^{>\delta}$ are identical to the ones for $\mathcal{D}_{M}^{=\delta}$, $\mathcal{D}_{M}^{>\delta}$, respectively, except for the set of transitions described in item 3 above. Thus, instead of transitions of the form $(x,(a,b),z>0,x,-1)$, for $a \neq b$, $\mathcal{D}_{gM}^{=\delta}$, $\mathcal{D}_{gM}^{>\delta}$ contain transitions of the form $(x,(a,b),z>0,x,-\texttt{gdiff}(a,b))$, for $a \neq b$, which read a pair (a,b) of distinct symbols, and decrement the counter z by the corresponding mismatch penalty $\texttt{gdiff}(a,b)$.

Note that the size of $\mathcal{D}_{gM}^{=\delta}$ or $\mathcal{D}_{gM}^{>\delta}$ is $\mathcal{O}(\delta+|\Sigma|^2\texttt{MAX}_{\texttt{gdiff}_\Sigma})$, where $\texttt{MAX}_{\texttt{gdiff}_\Sigma}$ is the maximum mismatch penalty over Σ.The following lemma states the correctness of the above automata constructions.

Lemma 9.4.2. *$\mathcal{D}_{gM}^{=\delta}$, $\mathcal{D}_{gM}^{>\delta}$ accept a pair of strings (s,t) iff $d_{gM}(s,t) = \delta$, $d_{gM}(s,t) > \delta$, respectively.*

9.5 Levenshtein Distance-Tracking Automaton

In [53], the authors show that for a given integer k, a relation $R \subseteq \Sigma^\star \times \Sigma^\star$ is rational if and only if for every $(s,t) \in R$, $|s| - |t| < k$. It is known from [44], that a subset is rational iff it is the behavior of a finite automaton. Thus, it follows from the above results that there exists a DFA that accepts the set of pairs of strings that are within bounded edit distance from each other. However, these theorems do not provide a constructive procedure for such an automaton. In what follows, we present novel constructions for DFA's $\mathcal{D}_{L}^{=\delta}$,

$\mathcal{D}_L^{>\delta}$ that accept pairs of strings (s,t) such that $d_L(s,t) = \delta$, $d_L(s,t) > \delta$, respectively.

The standard algorithm for computing the Levenshtein distance $d_L(s,t)$ is a dynamic programming-based algorithm using the recurrence relations in eq.(9.2). This algorithm organizes the bottom-up computation of the Levenshtein distance with the help of a table $\mathbf{t}$ of height $|s|$ and width $|t|$. The 0^{th} row and column of $\mathbf{t}$ account for the base case of the recursion. The $\mathbf{t}(i,j)$ entry stores the Levenshtein distance of the strings $s[0,i]$ and $t[0,j]$. In general, the entire table has to be populated in order to compute $d_L(s,t)$. However, when one is only interested in some bounded distance δ, then for every i, the algorithm only needs to compute values for the cells from $\mathbf{t}(i,i-\delta)$ to $\mathbf{t}(i,i+\delta)$ [62]. We call this region the δ-*diagonal* of $\mathbf{t}$, and use this observation to construct the DFA's $\mathcal{D}_L^{=\delta}$ and $\mathcal{D}_L^{>\delta}$.

We define $\mathcal{D}_L^{=\delta}$, $\mathcal{D}_L^{>\delta}$ to synchronously run on a pair of strings s, t, and accept iff $d_L(s,t) = \delta$, $d_L(s,t) > \delta$, respectively. In each step, $\mathcal{D}_L^{=\delta}$, $\mathcal{D}_L^{>\delta}$ read a pair of input symbols and change state to mimic the bottom-up edit distance computation by the dynamic programming algorithm.

Example Run. A run of $\mathcal{D}_L^{>2}$ on the string pair $s = accca$, $t = caca$, that checks if $d_L(s,t) > 2$, is shown in Fig. 9.1. After reading the i^{th} input symbol pair, $\mathcal{D}_L^{>2}$ uses its state to remember the last $\delta = 2$ symbols of s and t that it has read, and transitions to a state that contains the values of $\mathbf{t}(i,i)$ and the cells within the δ-diagonal, above and to the left of $\mathbf{t}(i,i)$.

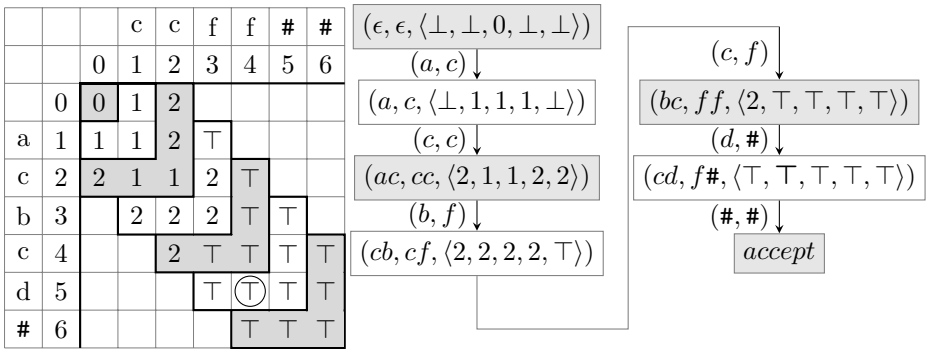


Figure 9.1: Dynamic programming table emulated by $\mathcal{D}_L^{>2}$. The table $\mathbf{t}$ filled by the dynamic programming algorithm is shown to the left, and a computation of $\mathcal{D}_L^{>2}$ on the strings $s = acbcd$ and $t = ccff$ is shown to the right.

As in the case of Manhattan distance, $\mathcal{D}_L^{=\delta}$, $\mathcal{D}_L^{>\delta}$ are identical, except for their accepting transitions. Formally, $\mathcal{D}_L^{=\delta}$, $\mathcal{D}_L^{>\delta}$ are each defined as a tuple $(\Sigma^{\#} \times \Sigma^{\#}, Q, q_0, \Delta, \{acc\})$, where $(\Sigma^{\#} \times \Sigma^{\#})$, Q, q_0, Δ, $\{acc\}$ are the input alphabet, the set of states, the initial state, the transition function and the singleton set of final states respectively. A state is defined as the tuple $(x, y, \mathbf{e})$, where x and y are strings of length at most δ and $\mathbf{e}$ is a vector containing $2\delta + 1$ entries, with values from $\{0, 1, \ldots, \delta, \bot, \top\}$ for each entry. A state of $\mathcal{D}_L^{=\delta}$, $\mathcal{D}_L^{>\delta}$ maintains the invariant that if i symbol pairs have been read, then x, y store the last δ symbols of s, t (i.e., $x = s[i\text{-}\delta\text{+}1, i]$, $y = t[i\text{-}\delta\text{+}1, i]$), and the entries in $\mathbf{e}$ correspond to the values stored in $\mathbf{t}(i, i)$ and the cells within the δ-diagonal, above and to the left of $\mathbf{t}(i, i)$ (i.e., entries in $\mathbf{e}$ correspond to the values in $\mathbf{t}(i, i)$ and in the cells defined by the sets $\{\mathbf{t}(i, j) \mid j \in [i\text{-}\delta, i\text{-}1]\}$, and $\{\mathbf{t}(j, i) \mid j \in [i\text{-}\delta, i\text{-}1]\}$. The values in these cells greater than δ are replaced by $\top$. The initial state is $q_0 = (\epsilon, \epsilon, \langle \bot, \ldots, \bot, 0, \bot, \ldots, \bot \rangle)$, where ϵ denotes

the empty string, $\perp$ is a special symbol denoting an undefined value, and the value 0 corresponds to entry $\mathbf{t}(0,0)$.

Upon reading the i^{th} input symbol pair, $s[i]$, $t[i]$, $\mathcal{D}_L^{=\delta}$, $\mathcal{D}_L^{>\delta}$ transition from state $q_{i\text{-}1} = (x_{i\text{-}1}, y_{i\text{-}1}, \mathbf{e}_{i\text{-}1})$ to a state $q_i = (x_i, y_i, \mathbf{e}_i)$ as shown in Fig. 9.2. First, note that x_i, y_i are the δ-length suffices of $x_{i\text{-}1}.s[i]$, $y_{i\text{-}1}.t[i]$, respectively. Also, note that to compute values in $\mathbf{e}_i$ corresponding to the i^{th} row of $\mathbf{t}$, we need the substring $t[i\text{-}\delta, i\text{-}1]$, the values $\mathbf{t}(i\text{-}1\text{-}\delta, i\text{-}1)$ to $\mathbf{t}(i\text{-}1, i\text{-}1)$, and the symbol $s[i]$. From the state invariant, it follows that the values of the required cells from $\mathbf{t}$ and the required substring $t[i\text{-}\delta, i\text{-}1]$ are present in $q_{i\text{-}1}$. Similarly, to compute $\mathbf{t}(j,i)$, where $j \in [i\text{-}1\text{-}\delta, i]$ the string in y_{i-1}, values in $\mathbf{e}_{i-1}$ and the input symbol suffice. Thus, given any state $q_{i\text{-}1}$ of $\mathcal{D}_L^{=\delta}$, $\mathcal{D}_L^{>\delta}$ and an input symbol pair, we can construct the unique next state that satisfies the state-invariant from $x_{i\text{-}1}$, $y_{i\text{-}1}$, $\mathbf{e}_{i\text{-}1}$ and the input symbol pair.

Finally, upon reading the symbol $(\#, \#)$ in state $(x, y, \mathbf{e})$, we add transitions to the single accepting state *acc* in $\mathcal{D}_L^{=\delta}$ (and in $\mathcal{D}_L^{>\delta}$) iff:

- $|s| = |t|$, i.e., x and y do not contain #, and the $(\delta+1)^{th}$ entry in $\mathbf{e}$ is δ ($\top$ in the case of $\mathcal{D}_L^{>\delta}$), or,
- $|s| = |t| + \ell$, i.e., y contains ℓ #'s, x contains no #, and the $(\delta+1-\ell)^{th}$ entry in $\mathbf{e}$ is δ ($\top$ in the case of $\mathcal{D}_L^{>\delta}$), or,
- $|t| = |s| + \ell$, i.e., x contains ℓ #'s, y contains no #, and the $(\delta+1+\ell)^{th}$ entry in $\mathbf{e}$ is δ ($\top$ in the case of $\mathcal{D}_L^{>\delta}$).

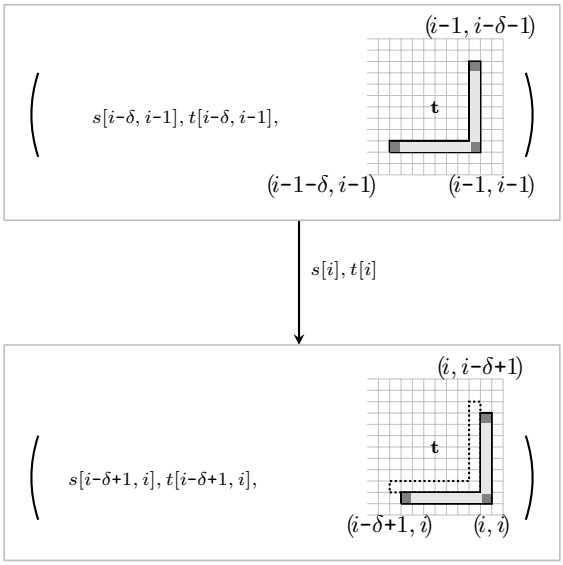


Figure 9.2: A transition of $\mathcal{D}_L^{=\delta}$, $\mathcal{D}_L^{>\delta}$

Upon reaching *acc*, $\mathcal{D}_L^{=\delta}$, $\mathcal{D}_L^{>\delta}$ remain in it for all possible input symbol pairs.

Note that the size of $\mathcal{D}_L^{=\delta}$ or $\mathcal{D}_L^{>\delta}$ is $\mathcal{O}((\delta|\Sigma|)^{4\delta})$. The following lemma states the correctness of these constructions. The proof follows from the state-invariants maintained by $\mathcal{D}_L^{=\delta}$, $\mathcal{D}_L^{>\delta}$ and their accepting transitions.

Lemma 9.5.1. *$\mathcal{D}_L^{=\delta}$, $\mathcal{D}_L^{>\delta}$ accept a pair of strings (s,t) iff $d_L(s,t) = \delta$, $d_L(s,t) > \delta$, respectively.*

Generalized Levenshtein Distance-Tracking Automata. The definitions for the automata $\mathcal{D}_{gL}^{=\delta}$, $\mathcal{D}_{gL}^{>\delta}$ that accept pairs of strings (s,t) such that $d_{gL}(s,t) = \delta$, $d_{gL}(s,t) > \delta$, respectively, with $d_{gL}(s,t)$ being the generalized Manhattan distance between s and t, are essentially the same as the definitions for $\mathcal{D}_L^{=\delta}$, $\mathcal{D}_L^{>\delta}$, respectively. The only difference lies in the fact that the transi-

tions of the form shown in Fig. 9.2 involve a computation using the mismatch and gap penalties, according to eq.(9.4).

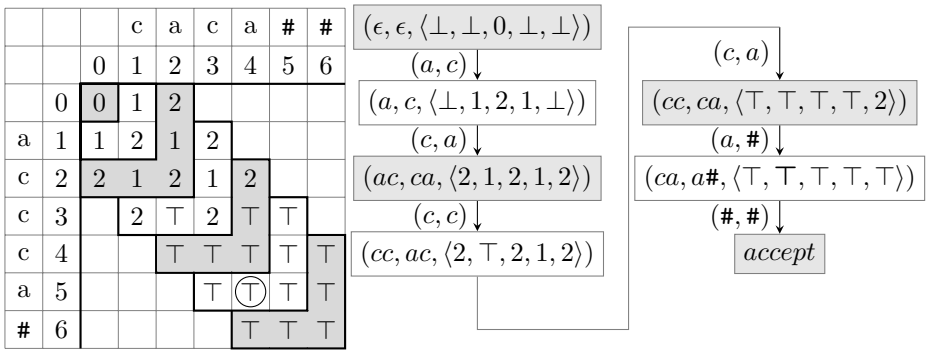


Figure 9.3: Dynamic programming table emulated by $\mathcal{D}_{gL}^{>2}$. The table $\mathbf{t}$ filled by the dynamic programming algorithm for $\delta = 2$ is shown to the left, and a computation of $\mathcal{D}_{L}^{>2}$ on the strings $s = accca$, $t = caca$ is shown to the right. Here, $\Sigma = \{a, b, c\}$, $\texttt{gdiff}(a, b) = \texttt{gdiff}(b, c) = \texttt{gdiff}(a, \#) = 1$, $\texttt{gdiff}(a, c) = \texttt{gdiff}(b, \#) = 2$, $\texttt{gdiff}(c, \#) = 3$ and $\alpha = 1$.

Example Run. A run of $\mathcal{D}_{gL}^{>2}$ on the string pair $s = accca$, $t = caca$ that checks if $d_{gL}(s, t) > 2$, is shown in Fig. 9.3. The mismatch and gap penalties are as enumerated in the caption.

Thus, the size of $\mathcal{D}_{gL}^{=\delta}$ or $\mathcal{D}_{gL}^{>\delta}$ is $\mathcal{O}((\delta|\Sigma|)^{4\delta})$, and the following lemma states the correctness of these constructions.

Lemma 9.5.2. *$\mathcal{D}_{gL}^{=\delta}$, $\mathcal{D}_{gL}^{>\delta}$ accept a pair of strings (s, t) iff $d_{gL}(s, t) = \delta$, $d_{gL}(s, t) > \delta$, respectively.*

Chapter 10

Robustness Analysis of String Transducers

Overview. In this chapter, we target robustness analysis of string transducers with respect to input perturbations. We formally define our notion of robustness and present a set of decision procedures based on reducing the problem of robustness verification of a transducer to the problem of checking the emptiness of a reversal-bounded counter machine. In this chapter, we choose to focus on the generalized Levenshtein and Manhattan distance metrics over strings. This is because string transducers are involved in a diverse set of applications, as highlighted in Chapter 1, and the usual Levenshtein and Manhattan distance metrics may be too coarse and inadequate for some of these applications.

10.1 Robust String Transducers

Our notion of robustness for finite transducers is an adaptation of the analytic notion of Lipschitz continuity. Given two metric spaces (M_1, d_1) and (M_2, d_2), a function $f : M_1 \to M_2$ is called *Lipschitz continuous* if there exists a constant $K \in \mathbb{N}$ such that $\forall x, y \in M_1 : d_2(f(x), f(y)) \leq K d_1(x, y)$. Intuitively, a Lipschitz-continuous function responds proportionally, and hence robustly, to

perturbations in its input. Thus, Lipschitz-continuity can be the basis of a mathematical definition of robustness [24].

We define a transducer $\mathcal{T}$ to be robust if the function $[\![\mathcal{T}]\!]$ encoded by the transducer satisfies a property very similar to Lipschitz-continuity. The one difference between the Lipschitz criterion and ours is that the output of a Lipschitz-continuous function changes proportionally to *every* perturbation δ to the input, however large δ might be. From the modeling point of view, this requirement seems too strong; if the input is noisy beyond a certain point, it makes little sense to constrain the behavior of the output. Accordingly, we define robustness of a transducer $\mathcal{T}$ with respect to a *fixed bound* B on the amount of input perturbation.

Definition 10.1.1 (Robust String Transducers)**.** Given an upper bound B on the input perturbation, a constant $K \in \mathbb{N}$ and a distance metric $d : \Sigma^* \times \Sigma^* \cup \Gamma^\star \times \Gamma^\star \rightarrow \mathbb{N}$, a transducer $\mathcal{T}$ defined over a regular language $\mathcal{L} \subseteq \Sigma^*$, with $[\![\mathcal{T}]\!] : \mathcal{L} \rightarrow \Gamma^\star$, is called (B, K)-robust if:

$$\forall \delta \leq B, \forall s, t \in \mathcal{L} : \; d(s,t) = \delta \implies d([\![\mathcal{T}]\!](s), [\![\mathcal{T}]\!](t)) \leq K\delta.$$

10.2 Robustness Analysis

From Def. 10.1.1, it follows that checking (B, K)-robustness of a transducer $\mathcal{T}$ is equivalent to checking if *for each* $\delta \leq B$, $\forall s, t \in L : d(s,t) = \delta \implies d([\![\mathcal{T}]\!](s), [\![\mathcal{T}]\!](t)) \leq K\delta$. Hence, we focus on the problem of checking robustness of a transducer for some *fixed input perturbation* δ. We reduce this

problem to checking language emptiness of a product machine $\mathcal{A}^{\delta}$ constructed from:

1. An input automaton $\mathcal{A}_I^{\delta}$ that accepts a pair of strings (s, t) iff $d(s, t) = \delta$,
2. A pair-transducer $\mathcal{P}$ that transforms input string pairs (s, t) to output string pairs (s', t') according to $\mathcal{T}$, and
3. An output automaton $\mathcal{A}_O^{\delta}$ that accepts (s', t') iff $d(s', t') > K\delta$.

We construct $\mathcal{A}^{\delta}$ such that $\mathcal{T}$ is robust iff for all $\delta \leq B$, the language of $\mathcal{A}^{\delta}$ is empty.

Later in this section, we present specialized constructions for $\mathcal{A}_I^{\delta}$, $\mathcal{A}_O^{\delta}$ for checking robustness of Mealy machines and functional transducers, with respect to the generalized Manhattan and Levenshtein distances. The definition of the pair-transducer $\mathcal{P}$ is standard in all these scenarios, and hence we present it first. We next define the product machine $\mathcal{A}^{\delta}$ for two relevant scenarios. Scenario 1 is when $\mathcal{A}_I^{\delta}$ and $\mathcal{A}_O^{\delta}$ are both DFAs - as we will see, this scenario presents itself while checking robustness of either type of transducer with respect to the generalized Levenshtein distance. Scenario 2 is when $\mathcal{A}_I^{\delta}$ and $\mathcal{A}_O^{\delta}$ are both 1-reversal-bounded counter machines - this scenario presents itself while checking robustness of either type of transducer with respect to the generalized Manhattan distance.

Recall that $\Sigma^{\#} = \Sigma \cup \{\#\}$. Let $\Gamma^{\#} = \Gamma \cup \{\#\}$, $\Gamma^{\epsilon,\#} = \Gamma \cup \{\epsilon, \#\}$, $\widetilde{\Sigma} = \Sigma^{\#} \times \Sigma^{\#}$ and $\widetilde{\Gamma} = \Gamma^{\epsilon,\#} \times \Gamma^{\epsilon,\#}$.

Pair-transducer, $\mathcal{P}$. Given a transducer $\mathcal{T}$, the pair-transducer $\mathcal{P}$ reads an input string pair and produces an output string pair according to $\mathcal{T}$. Formally, given $\mathcal{T} = (Q, \Sigma, \Gamma, q_0, E, F)$, $\mathcal{P}$ is defined as the tuple $(Q_{\mathcal{P}}, \widetilde{\Sigma}, \widetilde{\Gamma}, q_{0_{\mathcal{P}}}, E_{\mathcal{P}}, F_{\mathcal{P}})$ where $Q_{\mathcal{P}} = Q \times Q$, $q_{0_{\mathcal{P}}} = (q_0, q_0)$, $F_{\mathcal{P}} = F \times F$, and, $E_{\mathcal{P}}$ is the set of all transitions of the form $((q_1, q_2), (a, b), (w', v'), (q_1', q_2'))$ such that $(q_1, a, w', q_1') \in E$ and $(q_2, b, v', q_2') \in E$. While for Mealy machines, in all transitions in $E_{\mathcal{P}}$, w', v' are symbols in $\Gamma \cup \{\#\}$, for arbitrary functional transducers, w', v' may be strings of different lengths, and either or both could be ϵ. We define the function $[\![\mathcal{P}]\!]$ such that $[\![\mathcal{P}]\!](s,t) = (s',t')$ if $[\![\overset{]\!]}{\rightarrow}(s) = s'$ and $[\![\overset{]\!]}{\rightarrow}(t) =< t'$.

Product machine, $\mathcal{A}^{\delta}$. Given input automaton $\mathcal{A}_I^{\delta}$, pair transducer $\mathcal{P}$ and output automaton $\mathcal{A}_O^{\delta}$, the product machine $\mathcal{A}^{\delta}$ is constructed to accept all string pairs (s,t) such that (s,t) is accepted by $\mathcal{A}_I^{\delta}$ and there exists a string pair (s',t') accepted by $\mathcal{A}_O^{\delta}$ with $(s',t') = [\![\mathcal{P}]\!](s,t)$. Notice that while in each of its transitions, $\mathcal{A}_O^{\delta}$ can only read a pair of *symbols* at a time, each transition of $\mathcal{P}$ potentially generates a pair of (possibly unequal length) output *strings*. Hence, $\mathcal{A}^{\delta}$ cannot be constructed as a simple synchronized product.

Scenario 1. Given a DFA input automaton $\mathcal{A}_I^{\delta} = (Q_I, \widetilde{\Sigma}, q_{0_I}, \Delta_I, F_I)$, pair transducer $\mathcal{P} = (Q_{\mathcal{P}}, \widetilde{\Sigma}, \widetilde{\Gamma}, q_{0_{\mathcal{P}}}, E_{\mathcal{P}}, F_{\mathcal{P}})$ and a DFA output automaton $\mathcal{A}_O^{\delta} = (Q_O, \widetilde{\Gamma}, q_{0_O}, \Delta_O, F_O)$, $\mathcal{A}^{\delta}$ is a DFA given by the tuple $(Q, \widetilde{\Sigma}, q_0, \Delta, F)$, where $Q \subseteq Q_I \times Q_{\mathcal{P}} \times Q_O$, $q_0 = (q_{0_I}, q_{0_{\mathcal{P}}}, q_{0_O})$, $F = F_I \times F_{\mathcal{P}} \times F_O$, and E is defined as follows:

$\Delta((q_I, q_{\mathcal{P}}, q_O), (a, b)) = (q'_I, q'_{\mathcal{P}}, q'_O)$ iff

1. $\Delta_I(q_I, (a, b)) = q'_I$, and
2. there exist w', v' such that
   (a) $(q_{\mathcal{P}}, (a, b), (w', v'), q'_{\mathcal{P}}) \in E_{\mathcal{P}}$, and
   (b) $\Delta^*_O(q_O, (w', v')) = q'_O$.

Scenario 2. For counter machines, one also needs to keep track of the counters. Given input automaton $\mathcal{A}^\delta_I = (\widetilde{\Sigma}, X_I, x_{0_I}, Z_I, G_I, E_I, F_I)$ in NCM(h_I,1), pair transducer $\mathcal{P} = (Q_{\mathcal{P}}, \widetilde{\Sigma}, \widetilde{\Gamma}, q_{0_{\mathcal{P}}}, E_{\mathcal{P}}, F_{\mathcal{P}})$ and output automaton $\mathcal{A}^\delta_O$ in class NCM(h_O,1), of the form $(\widetilde{\Gamma}, X_O, x_{0_O}, Z_O, G_O, E_O, F_O)$, $\mathcal{A}^\delta$ is in NCM(h,1), with $h = h_I + h_O$, and is given by the tuple $(\widetilde{\Sigma}, X, x_0, Z, G, E, F)$, where $X \subseteq X_I \times Q_{\mathcal{P}} \times X_O$, $x_0 = (x_{0_I}, q_{0_{\mathcal{P}}}, x_{0_O})$, $Z = Z_I \cup Z_O$, $G = G_I \cup G_O$, $F = F_I \times F_{\mathcal{P}} \times F_O$, and E is defined as follows:
$((x_I, q_{\mathcal{P}}, x_O), (a, b), t, (x'_I, q'_{\mathcal{P}}, x'_O), g_{I1}, \ldots, g_{Ih_I}, g_{O1}, \ldots, g_{Oh_O}) \in E$ iff

1. $(x_I, (a, b), t_I, x'_I, g_{I1}, \ldots, g_{Ih_I}) \in E_I$ with $t \Rightarrow t_I$, and
2. there exist w', v' such that
   (a) $(q_{\mathcal{P}}, (a, b), (w', v'), q'_{\mathcal{P}}) \in E_{\mathcal{P}}$, and
   (b) $(x_O, (w'[1], v'[1]), z_{O1}, \ldots, z_{Oh_O}) \rightarrow^\star_{\mathcal{A}^\delta_O} (x'_O, (w'[j], v'[\ell]), z'_{O1}, \ldots, z'_{Oh_O})$, with $j = |w'|$, $\ell = |v'|$, $t \Rightarrow t_O$ where t_O is the enabling test corresponding to the first move along $\rightarrow^\star_{\mathcal{A}^\delta_O}$ and $\forall k$: $z'_{Ok} = z_{Ok} + g_{Ok}$.

Thus, in both scenarios, each transition of $\mathcal{A}^\delta$ corresponds to a single transition of $\mathcal{A}_I^\delta$, a single transition of $\mathcal{P}$ yielding some output string pair (w', v'), and a path of $\mathcal{A}_O^\delta$, consisting of multiple transitions/moves on (w', v').

10.2.1 Mealy Machines

Generalized Manhattan Distance. For a Mealy machine $\mathcal{T}$, it is easy to see from the descriptions of $\mathcal{A}_I^\delta$, $\mathcal{A}_O^\delta$ and from the constructions in Sec. 9.4, that $\mathcal{A}_I^\delta$ is the same as $\mathcal{D}_{gM}^{=\delta}$ and $\mathcal{A}_O^\delta$ is essentially the same as $\mathcal{D}_{gM}^{>K\delta}$, with the alphabet being $\widetilde{\Gamma}$. Thus, $\mathcal{A}_I^\delta$ and $\mathcal{A}_O^\delta$ are both in NCM(1,1). Let $\mathcal{A}^\delta$ be the product machine, as defined in *Scenario 2* using $\mathcal{A}_I^\delta$, $\mathcal{P}$ and $\mathcal{A}_O^\delta$. From Lem. 9.4.2 and the definition of $\mathcal{A}^\delta$, it follows that $\mathcal{A}^\delta$ accepts all input strings (s, t) such that $d_M(s, t) = \delta$, and there exists $(s', t') = [\![\mathcal{P}]\!](s, t)$ with $d_M(s', t') > K\delta$. Thus, any pair of input strings accepted by $\mathcal{A}^\delta$ is a witness to the non-robustness of $\mathcal{T}$; equivalently $\mathcal{T}$ is robust iff $\mathcal{A}^\delta$ is empty for all $\delta \leq B$.

The product machine $\mathcal{A}^\delta$ is in NCM(2, 1) and its size is polynomial in $size(\mathcal{T})$, δ, K, $|\Sigma|$, $|\Gamma|$ and $\texttt{MAX}_{\texttt{diff}}$, where $\texttt{MAX}_{\texttt{diff}}$ is the maximum mismatch penalty over Σ and Γ. Since, we need to check nonemptiness of $\mathcal{A}^\delta$ for all $\delta \leq B$, we have the following theorem using Lem. 9.3.1.

Theorem 10.2.1. *Robustness verification of a Mealy machine $\mathcal{T}$ with respect to the generalized Manhattan distance can be accomplished in* NLOGSPACE *in $size(\mathcal{T})$, B, K, $|\Sigma|$, $|\Gamma|$ and $\texttt{MAX}_{\texttt{diff}}$ (maximum mismatch penalty).*

Generalized Levenshtein Distance. For a Mealy machine $\mathcal{T}$, $\mathcal{A}_I^\delta$ is the

same as $\mathcal{D}_{gL}^{=\delta}$ and $\mathcal{A}_O^\delta$ is the same as $\mathcal{D}_{gL}^{>K\delta}$ (as defined in Sec. 9.4), with alphabet $\widetilde{\Gamma}$. Thus, $\mathcal{A}_I^\delta$ and $\mathcal{A}_O^\delta$ are both DFAs. Let $\mathcal{A}^\delta$ be a product machine, as defined in *Scenario 1* using $\mathcal{A}_I^\delta$, $\mathcal{P}$ and $\mathcal{A}_O^\delta$. As before, from Lem. 9.5.2 and the definition of $\mathcal{A}^\delta$, it follows that $\mathcal{T}$ is robust iff $\mathcal{A}^\delta$ is empty for all $\delta \leq B$.

The size of $\mathcal{A}^\delta$ is $\mathcal{O}(size^2(\mathcal{T})|\Sigma|^{4\delta}(|\Gamma|K)^{4K\delta}\delta^{4\delta(1+K)})$. Since the emptiness of the DFA $\mathcal{A}^\delta$ can be checked in NLOGSPACE in the size of $\mathcal{A}^\delta$, and we need to repeat this for all $\delta \leq B$, we have the following theorem.

Theorem 10.2.2. *Robustness verification of a Mealy machine $\mathcal{T}$ with respect to the generalized Levenshtein distance can be accomplished in* PSPACE *in B and K.*

10.2.2 Functional Transducers

Checking robustness of functional transducers is more involved than checking robustness of Mealy machines. The main reason is that $\mathcal{P}$ may produce output symbols for two strings in an unsynchronized fashion, i.e., the symbols read by $\mathcal{A}_O^\delta$ may be of the form (a, ϵ) or (ϵ, a). While this does not affect the input automata constructions, the output automata for functional transducers differ from the ones for Mealy machines.

Generalized Manhattan Distance. As stated above, $\mathcal{A}_I^\delta$ is the same as $\mathcal{D}_{gM}^{=\delta}$. The construction of $\mathcal{A}_O^\delta$ is based on the observation that if s', t' are mismatched in $1 + K\delta$ positions, $d_{gM}(s', t')$ is guaranteed to be greater than $K\delta$. Let $\eta = 1 + K\delta$. We define $\mathcal{A}_O^\delta$ to be in class NCM$(1 + 2\eta, 1)$ with a

distance counter z and two sets of *position counters* $c_1, \ldots, c_\eta$ and $d_1, \ldots, d_\eta$. The counter z is initialized to $K\delta$ and for all j, position counters c_j, d_j are initialized to hold guesses for η *mismatch positions* in s', t', respectively. In particular, the position counters are initialized such that for all j, $c_j = d_j$, $c_j \geq 0$, and $c_j < c_{j+1}$, thereby ensuring that the counter pairs store η *distinct* position guesses [1]. For notational convenience, we denote the initial position guess stored in the position counter c_j (or d_j) by p_j.

Intuitively, for each j, $\mathcal{A}_O^\delta$ uses its position counters to compare the symbols at the p_j^{th} position of each string. For all j, $\mathcal{A}_O^\delta$ decrements c_j, d_j upon reading a nonempty symbol of s', t', respectively. Thus, $\mathcal{A}_O^\delta$ reads the p_j^{th} symbol of s', t' when $c_j = 0$, $d_j = 0$, respectively. If the p_j^{th} symbols are mismatched symbols a, b, then $\mathcal{A}_O^\delta$ decrements the distance counter z by $\texttt{gdiff}(a, b)$. Now, recall that the symbol at the p_j^{th} position for one string may arrive before that for the other string. Thus, for instance, c_j may be 0, while d_j is still positive. In this case, $\mathcal{A}_O^\delta$ needs to remember the *earlier* symbol in its state till the *delayed* symbol arrives. Note that $\mathcal{A}_O^\delta$ has to remember at most η symbols corresponding to the η guessed positions. When the delayed symbol at position p_j of the trailing string arrives, i.e. d_j finally becomes 0, $\mathcal{A}_O^\delta$ compares it to the symbol stored in its state and decrements z as needed.

[1] Note that this can be done nondeterministically as follows. First all 2η counters are incremented by 1, and at some nondeterministically chosen point, the machine stops incrementing the c_1, d_1 counters, then at some further point stops incrementing the c_2, d_2 counters, and so on. This ensures that for each j, $c_j = d_j$, and the higher index counters have higher (distinct) values.

Formally, a state of $\mathcal{A}_O^\delta$ is a tuple of the form (pos, id, vec), where $pos \in [1, \eta]$ is a positive integer (initially 0) that keeps track of the earliest position for which $\mathcal{A}_O^\delta$ is waiting to read symbols of both strings, $id \in \{0, 1, 2\}$ is used to track which of the strings is leading the other, and vec is a η-length vector that stores the symbols of the leading string. Initially, all entries of vec are $\perp$. The invariant maintained by the state is as follows: if $pos = j$, (a) $id = 0$ iff $c_j > 0$, $d_j > 0$ and $vec_j = \perp$, (b) $id = 1$ iff $c_j \leq 0$, $d_j > 0$ and $vec_j = s'[p_j]$, and (c) $id = 2$ iff $c_j > 0$, $d_j \leq 0$ and $vec_j = t'[p_j]$. Thus, if c_j becomes zero while d_j is non-zero, id is set to 1, and vec_j is set to the symbol read, $s'[p_j]$; when d_j eventually becomes zero due to the p_j^{th} symbol of t' being read, then vec_j is set to $\perp$, z is decremented by $\texttt{gdiff}(s'[p_j], t'[p_j])$ and pos is incremented. The case when the p_j^{th} symbol of t' is output before that of s' is handled symmetrically. $\mathcal{A}_O^\delta$ moves to an accepting state whenever the value in z goes below 0, i.e. $d_{gM}(s', t') > K\delta$, and stays there. $\mathcal{A}_O^\delta$ moves to a special rejecting state if the value in z is nonnegative and either the string pairs or all position guesses are exhausted, i.e., if $\mathcal{A}_O^\delta$ reads a $(\#, \#)$ symbol or if all position counters are negative. In effect, the construction ensures that if $\mathcal{A}_O^\delta$ accepts a pair of strings (s', t'), then $d_{gM}(s', t') > K\delta$. On the other hand, note that if $d_{gM}(s', t') > K\delta$, then there exists a run of $\mathcal{A}_O^\delta$ in which it correctly guesses some mismatch positions (whose number is at most η) such that their cumulative mismatch penalty exceeds $K\delta$.

Lemma 10.2.3. *The above $\mathcal{A}_O^\delta$ accepts a pair of strings (s, t) iff $d_{gM}(s, t) > K\delta$.*

Note that the size of $\mathcal{A}_O^\delta$ is $\mathcal{O}(\Gamma^{2K\delta})$. Let $\mathcal{A}^\delta$ be a product machine, as defined in *Scenario 2* using $\mathcal{A}_I^\delta$, $\mathcal{P}$ and $\mathcal{A}_O^\delta$. From Lem. 9.4.2, Lem. 10.2.3 and the definition of $\mathcal{A}^\delta$, it follows that $\mathcal{T}$ is robust iff $\mathcal{A}^\delta$ is empty for all $\delta \leq B$. $\mathcal{A}^\delta$ is in class NCM$(2 + 2\eta, 1)$, and its size is $\mathcal{O}(size^2(\mathcal{T})(\delta + |\Sigma|^2\mathtt{MAX}_{\mathtt{diff}_\Sigma})\Gamma^{2K\delta})$, with $\mathtt{MAX}_{\mathtt{diff}}$ being the maximum mismatch penalty over Σ. Since we need to repeat this for all $\delta \leq B$, we have the following theorem using Lem. 9.3.1.

Theorem 10.2.4. *Robustness verification of a functional transducer $\mathcal{T}$ with respect to the generalized Manhattan distance can be accomplished in* PSPACE *in B and K.*

Generalized Levenshtein distance. The input automaton $\mathcal{A}_I^\delta$ is the same as $\mathcal{D}_{gL}^{=\delta}$. In order to track the generalized Levenshtein distance between the unsynchronized output strings generated by $\mathcal{P}$, $\mathcal{A}_O^\delta$ needs to remember substrings of the leading string in its state, and not simply the symbols at possible mismatch positions. A natural question to ask is whether there exists a bound on the length of the substrings that $\mathcal{A}_O^\delta$ needs to remember in its state. We first address this question before defining $\mathcal{A}_O^\delta$.

Consider $\mathcal{A}_I^\delta \otimes \mathcal{P}$, the synchronous product of the input automaton $\mathcal{A}_I^\delta$ and the pair transducer $\mathcal{P}$. Let $\mathcal{T}_{I\otimes\mathcal{P}} = (Q_{I\otimes\mathcal{P}}, \widetilde{\Sigma}, \widetilde{\Gamma}, q_{0_{I\otimes\mathcal{P}}}, E_{I\otimes\mathcal{P}}, F_{I\otimes\mathcal{P}})$ be obtained by *trimming* $\mathcal{A}_I^\delta \otimes \mathcal{P}$, i.e., by removing all states that are not reachable from the initial state or from which no final state is reachable. The set $E_{I\otimes\mathcal{P}}$ of transitions of $\mathcal{T}_{I\otimes\mathcal{P}}$ can be extended in a natural way to the set $E^*_{I\otimes\mathcal{P}}$ of paths of $\mathcal{T}_{I\otimes\mathcal{P}}$. Note that for any path $(q_{0_{I\otimes\mathcal{P}}}, (w, v), (w', v'), q_{f_{I\otimes\mathcal{P}}})$ from the initial state to some final state $q_{f_{I\otimes\mathcal{P}}} \in F_{I\otimes\mathcal{P}}$, $d_{gL}(w, v) = \delta$ and $[\![\mathcal{P}]\!](w, v) = (w', v')$.

We define the *pairwise-delay* of a path π of $\mathcal{T}_{I\otimes\mathcal{P}}$, denoted $pd(\pi)$, as the difference in lengths of its output string labels: for $\pi = (q, (w, v), (w', v'), q')$, $pd(\pi) = \mathsf{abs}\,(|w'| - |v'|)$. $\mathcal{T}_{I\otimes\mathcal{P}}$ is said to have *bounded pairwise-delay* if the pairwise-delay of all its paths is bounded. For $\mathcal{T}_{I\otimes\mathcal{P}}$ with bounded pairwise-delay, we denote the maximum pairwise-delay over all paths of $\mathcal{T}_{I\otimes\mathcal{P}}$ by $\mathcal{D}(\mathcal{T}_{I\otimes\mathcal{P}})$. Let ℓ_{max} be the length of the longest output string in any transition of $\mathcal{T}$, i.e., $\ell_{max} = \max\{|w'| \mid (q, a, w', q') \in E\}$, and let Q_I, Q be the set of states of $\mathcal{A}_I^\delta$, $\mathcal{T}$.

Lemma 10.2.5. *$\mathcal{T}_{I\otimes\mathcal{P}}$ has bounded pairwise-delay, with $\mathcal{D}(\mathcal{T}_{I\otimes\mathcal{P}})$ bounded by $|Q|^2|Q_I|\ell_{max}$, iff the pairwise-delay of all cyclic paths in $\mathcal{T}_{I\otimes\mathcal{P}}$ is 0.*

Proof. If there is a cyclic path $c = (q, (w, v), (w', v'), q)$ in $\mathcal{T}_{I\otimes\mathcal{P}}$ with $pd(c) \neq 0$, then for n traversals through c, $pd(c^n) = n(pd(c))$, and hence $\mathcal{D}(\mathcal{T}_{I\otimes\mathcal{P}})$ is not bounded. If for all cycles c, $pd(c) = 0$, then for any path π, $pd(\pi) = pd(\pi_{acy})$, where π_{acy} is the acyclic path obtained from π by iteratively removing all cycles from π. Thus, $\mathcal{D}(\mathcal{T}_{I\otimes\mathcal{P}})$ is bounded by the maximum possible pairwise-delay along any acyclic path of $\mathcal{T}_{I\otimes\mathcal{P}}$. This maximum delay is $(|Q_{I\otimes\mathcal{P}}| - 1)\ell_{max}$ and is exhibited along an acyclic path of maximum length $|Q_{I\otimes\mathcal{P}}| - 1$, with the output string pair along each transition being ϵ and a string of length ℓ_{max}. By definition of $\mathcal{T}_{I\otimes\mathcal{P}}$, $|Q_{I\otimes\mathcal{P}}| \leq |Q|^2.|Q_I|$. The result follows. □

Corollary 10.2.6. *$\mathcal{T}_{I\otimes\mathcal{P}}$ has bounded pairwise-delay iff each simple cycle of $\mathcal{T}_{I\otimes\mathcal{P}}$ is labeled with equal length output strings.*

Lemma 10.2.7. *If $\mathcal{T}_{I\otimes\mathcal{P}}$ does not have bounded pairwise-delay, $\mathcal{T}$ is non-robust.*

Proof. We exhibit a witness for non-robustness of $\mathcal{T}$. If $\mathcal{T}_{I\otimes\mathcal{P}}$ does not have bounded pairwise-delay, there is some simple cycle $c : (q, (w_c, v_c), (w'_c, v'_c), q)$ in $\mathcal{T}_{I\otimes\mathcal{P}}$ with $|w'_c| \neq |v'_c|$. Consider the paths $\pi_1 = (q_{0_{I\otimes\mathcal{P}}}, (w_1, v_1), (w'_1, v'_1), q)$ and $\pi_2 = (q, (w_2, v_2), (w'_2, v'_2), q_{f_{I\otimes\mathcal{P}}})$, with $q_{f_{I\otimes\mathcal{P}}} \in F_{I\otimes\mathcal{P}}$. Let us assume that $|w'_1| > |v'_1|$, $|w'_c| > |v'_c|$ and $|w'_2| > |v'_2|$ (the other cases can be handled similarly). Let $|w'_c| - |v'_c| = l_c$ and $|w'_1.w'_2| - |v'_1.v'_2| = l$.

Then, given δ, K, there exists $n \in \mathbb{N}$ such that $l + nl_c > K\delta$. The witness path π to non-robustness of $\mathcal{T}$ can now be constructed from π_1, followed by n-traversals of c, followed by π_2. By definition of $\mathcal{T}_{I\otimes\mathcal{P}}$, the generalized Levenshtein distance, $d_{gL}(w_1.(w_c)^n.w_2, v_1.(v_c)^n.v_2)$, of the input string labels of π, equals δ, and by construction of π, the difference in the lengths, and hence the generalized Levenshtein distance, $gd_L(w'_1.(w'_c)^n.w'_2, v'_1.(v'_c)^n.v'_2)$ of the output string labels of π exceeds $K\delta$. □

Lem. 10.2.5 is helpful in constructing an output automaton $\mathcal{A}^{\delta}_{\mathcal{O}}$ that accepts a pair of output strings (s', t') iff $d_{gL}(s', t') > K\delta$. The construction of $\mathcal{A}^{\delta}_{\mathcal{O}}$ is very similar to that of $\mathcal{D}^{K\delta}_{gL}$, defined over alphabet $\widetilde{\Gamma}$, with one crucial difference. Having read the j^{th} symbol of s', in order to compute all entries in the j^{th} row of the $K\delta$-diagonal in the dynamic programming table, we need to have seen the $(j + K\delta)^{th}$ symbol of t'. However, the maximum delay between s' and t' could be as much as $\mathcal{D}(\mathcal{T}_{I\otimes\mathcal{P}})$ (by Lem. 10.2.5). Hence,

unlike $\mathcal{D}_{gL}^{K\delta}$, which only needs to remember strings of length $K\delta$ in its state, $\mathcal{A}_O^\delta$ needs to remember strings of length $\mathcal{D}(\mathcal{T}_{I\otimes\mathcal{P}}) + K\delta$ in its state. Thus, a state of $\mathcal{A}_O^\delta$ is a tuple $(x, y, \mathbf{e})$, where x and y are strings of length at most $\mathcal{D}(\mathcal{T}_{I\otimes\mathcal{P}}) + K\delta$, and $\mathbf{e}$ is a vector of length $2K\delta + 1$. Note that $\mathcal{A}_O^\delta$ is a DFA with size $\mathcal{O}(|\Gamma|^{4(K\delta+\mathcal{D}(\mathcal{T}_{I\otimes\mathcal{P}}))})$, where $\mathcal{D}(\mathcal{T}_{I\otimes\mathcal{P}})$ is the maximum pairwise-delay of $\mathcal{T}$ and is $\mathcal{O}(size^2(\mathcal{T})|\Sigma|^{4\delta}\delta^{4\delta}\ell_{max})$.

Lemma 10.2.8. *If $\mathcal{T}_{I\otimes\mathcal{P}}$ has bounded pairwise-delay, $\mathcal{A}_O^\delta$ as described above accepts a pair of strings (s', t') iff $d_{gL}(s', t') > K\delta$.*

Summarizing our robustness checking algorithm for a functional transducer $\mathcal{T}$, we first check if $\mathcal{T}_{I\otimes\mathcal{P}}$ does not have bounded pairwise-delay. To do this, we check if there exists a simple cycle c in $\mathcal{T}_{I\otimes\mathcal{P}}$ for which $pd(c) \neq 0$. If yes, $\mathcal{T}$ is non-robust by Lem. 10.2.7. If not, we construct the product machine $\mathcal{A}^\delta$, as defined in *Scenario 1* using $\mathcal{A}_I^\delta$, $\mathcal{P}$ and $\mathcal{A}_O^\delta$. By Lem. 9.5.2, Lem. **??** and the definition of $\mathcal{A}^\delta$, it follows that $\mathcal{T}$, with bounded pairwise-delay, is robust iff $\mathcal{A}^\delta$ is empty for all $\delta \leq B$.

Checking if there exists a simple cycle c in $\mathcal{T}_{I\otimes\mathcal{P}}$ with $pd(c) \neq 0$ can be done in NLOGSPACE in the size of $\mathcal{T}_{I\otimes\mathcal{P}}$[2], which is $\mathcal{O}(size^2(\mathcal{T})|\Sigma|^{4\delta}\delta^{4\delta})$. Also, the nonemptiness of $\mathcal{A}^\delta$ can be checked in NLOGSPACE in its size, as given by the product of $size(\mathcal{T}_{I\otimes\mathcal{P}})$ and $size(\mathcal{A}_O^\delta)$. Repeating this for all $\delta \leq B$, we have the following theorem.

[2]This can be done using a technique similar to the one presented in [122](Theorem 2.4) for checking nonemptiness of a Büchi automaton.

Theorem 10.2.9. *Robustness verification of a functional transducer $\mathcal{T}$ with respect to the Levenshtein distance can be accomplished in* ExpSpace *in* B.

Chapter 11

Robustness Analysis of Networked Systems

Overview. In this chapter, we target robustness analysis of networked systems with respect to perturbations in the network channels. We formally define our model for synchronous networked systems and our notion of robustness for such systems. We present automata-theoretic decision procedures for checking robustness with respect to the Levenshtein and Manhattan distance metrics.

In what follows, we sometimes denote vectors of objects using bold letters such as $\mathbf{s}$, $\boldsymbol{\varepsilon}$, with the i^{th} object in the vector denoted s_i, ε_i respectively.

11.1 Robust Networked Systems

In this section, we present a formal model for a synchronous networked system. We then introduce a notion of robustness for computations of such networked systems when the communication channels are prone to errors.

11.1.1 Synchronous Networked System

A networked system, denoted $\mathcal{N}$, can be described as a directed graph $(\mathcal{P}, \mathcal{C})$, with a set of processes $\mathcal{P} = \{P_1, \ldots, P_n\}$ and a set of communication channels

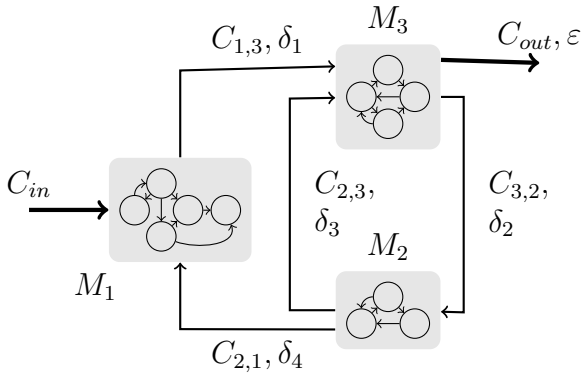


Figure 11.1: Networked system

$\mathcal{C}$. The set of channels consists of internal channels N, external input channels I, and external output channels O. An internal channel $C_{ij} \in N$ connects a source process P_i to a destination process P_j. An input channel has no source process, and an output channel has no destination process in $\mathcal{N}$.

Process Definition and Semantics. A process P_i in the networked system is defined as a tuple (In_i, Out_i, M_i), where $In_i \subseteq (I \cup N)$ is the set of P_i's input channels, $Out_i \subseteq (O \cup N)$ is the set of P_i's output channels, and M_i is a machine describing P_i's input/output behavior. We assume a synchronous model of computation: (1) at each tick of the system, each process consumes an input symbol from each of its input channels, and produces an output symbol on each its output channels, and (2) message delivery through the channels is instantaneous. We further assume that a networked system $\mathcal{N}$ has a computation alphabet Σ for describing the inputs and outputs of each process, and for describing communication over the channels. Please see Fig. 11.1 for

an example networked system. Observe that a process may communicate with one, many or all processes in $\mathcal{N}$ using its output channels. Thus our network model subsumes unicast, multicast and broadcast communication schemes.

In this work, we focus on processes described as Mealy machines. In each tick, a Mealy machine process in a networked system $\mathcal{N}$ consumes a composite symbol (the tuple of symbols on its input channels), and outputs a composite symbol (the tuple of symbols on its output channels). Thus, the input alphabet for $\mathcal{M}i$ is $\Sigma^{|In_i|}$, and the output alphabet is $\Sigma^{|Out_i|}$. Let $(Q_i, \Sigma^{|In_i|}, \Sigma^{|Out_i|}, , q_{0_i}, E_i)$ be the tuple describing the Mealy machine underlying process P_i. Recall from Sec. 9.1 that every state of a Mealy machine is an accepting state; hence, we do not include a set F of accepting states in the tuple describing a Mealy machine.

Operational Semantics of a Network. We define a *network state* $\mathbf{q}$ as the tuple $(q_1, \ldots, q_n, c_1, \ldots, c_{|N|})$, where for each i, $q_i \in Q_i$ is the state of P_i, and for each k, c_k is the state of the k^{th} internal channel, i.e., the current symbol in the channel. A transition of $\mathcal{N}$ has the following form:

$$\begin{array}{l} (q_1, \ldots, q_n, c_1, \ldots, c_{|N|}) \\ \qquad\qquad \Big\downarrow (a_1, \ldots, a_{|I|}), (a'_1, \ldots, a'_{|O|}) \\ (q'_1, \ldots, q'_n, c'_1, \ldots, c'_{|N|}) \end{array}$$

Here $(a_1, \ldots, a_{|I|})$ denote the symbols on the external input channels, and $(a'_1, \ldots, a'_{|O|})$ denote the symbols on the external output channels. During a

transition of $\mathcal{N}$, each process P_i consumes a composite symbol (given by the states of all internal channels in In_i and the symbols in the external input channels in In_i), changes state from q_i to q_i', and outputs a composite symbol. The generation of an output symbol by P_i causes an update to the states of all internal channels in Out_i and results in the output of a symbol on each output channel in Out_i.

Thus, we can view the networked system $\mathcal{N}$ itself as a machine that in each step, consumes an $|I|$-dimensional input symbol $\mathbf{a}$ from its external input channels, changes state according to the transition functions E_i of each process, and outputs an $|O|$-dimensional output symbol $\mathbf{a}'$ on its external output channels.

Formally, we define the semantics of a computation of $\mathcal{N}$ using the tuple $(\Sigma^{|I|}, \Sigma^{|O|}, Q, \mathbf{q}_0, E)$, where $Q = (Q_1 \times \ldots \times Q_n \times \Sigma^{|N|})$ is the set of states and $E \subseteq (Q \times \Sigma^{|I|} \times \Sigma^{|O|} \times Q)$ is the network transition function. The initial state $\mathbf{q}_o = (q_{0_1}, \ldots, q_{0_n}, c_{0_1}, \ldots, c_{0_{|N|}})$ of $\mathcal{N}$ is given by the initial process states and internal channel states. An execution $\rho(\mathbf{s})$ of $\mathcal{N}$ on an input string $\mathbf{s} = \mathbf{s}[1]\mathbf{s}[2]\ldots\mathbf{s}[m]$ is defined as a sequence of configurations of the form $(\mathbf{q}_0, \epsilon)$, $(\mathbf{q}_1, \mathbf{s}'[1])$, $\ldots$, $(\mathbf{q}_m, \mathbf{s}'[m])$, where for each j, $1 \leq j \leq m$, $(\mathbf{q}_{j-1}, \mathbf{s}[j], \mathbf{s}'[j], \mathbf{q}_j) \in E$. The output function computed by the networked system $[\![\mathcal{N}]\!] : (\Sigma^{|I|})^\star \rightarrow (\Sigma^{|O|})^\star$ is then defined such that $[\![\mathcal{N}]\!](\mathbf{s}[1]\mathbf{s}[2]\ldots\mathbf{s}[m]) = \mathbf{s}'[1]\mathbf{s}'[2]\ldots\mathbf{s}'[m]$.

11.1.2 Channel Perturbations and Robustness

An execution of a networked system is said to be perturbed if one or more of the internal channels are perturbed one or more times during the execution. A channel perturbation can be modeled as a deletion or substitution of the current symbol in the channel. To model symbol deletions[1], we extend the alphabet of each internal channel to $\Sigma^\epsilon = \Sigma \cup \epsilon$. A perturbed execution includes transitions corresponding to channel perturbations, of the form:

$$\begin{array}{c}(q_1, \ldots, q_n, c_1, \ldots, c_{|N|}) \\ \Big\downarrow \boldsymbol{\epsilon}, \boldsymbol{\epsilon} \\ (q'_1, \ldots, q'_n, c'_1, \ldots, c'_{|N|}),\end{array}$$

Here, for each i, the states q'_i and q_i are identical, and for some k, $c_k \neq c'_k$. Such transitions, termed τ-transitions[2], do not consume any input symbol and model instantaneous channel errors. We say that the k^{th} internal channel is perturbed in a τ-transition if $c_k \neq c'_k$. A perturbed network execution $\rho_\tau(\mathbf{s})$ on an input string $\mathbf{s} = \mathbf{s}[1]\mathbf{s}[2]\ldots\mathbf{s}[m]$ is a sequence of configurations $(\mathbf{q}_0, \boldsymbol{\epsilon}), \ldots, (\mathbf{q}_\tau, \mathbf{s}'[m])$, where for each j either $(\mathbf{q}_{j-1}, \mathbf{s}[\ell], \mathbf{s}'[\ell], \mathbf{q}_j) \in E$, for some ℓ, or $(\mathbf{q}_{j-1}, \boldsymbol{\epsilon}, \boldsymbol{\epsilon}, \mathbf{q}_j)$ is a τ-transition.

[1]Note that though a perturbation can cause a symbol on an internal channel to get deleted in a given step, we expect that the processes reading from this channel will output a nonempty symbol in that step. In this sense, we treat an empty input symbol simply as a special symbol, and assume that each process can handle such a symbol.

[2]Note that a network transition of the form $((q_1, \ldots, q_n, c_1, \ldots, c_{|N|}), \boldsymbol{\epsilon}, \mathbf{a}', (q'_1, \ldots, q'_n, c'_1, \ldots, c'_{|N|}))$ where for some i, $q_i \neq q'_i$ is not considered a τ-transition: such a transition involves a state change by some process on an empty input symbol along with the generation of a nonempty output symbol.

Note that there can be several possible perturbed executions of $\mathcal{N}$ on a string $\mathbf{s}$ which differ in their exact instances of τ-transitions and the channels perturbed in each instance. Each such perturbed execution generates a different perturbed output. For a specific perturbed execution $\rho_\tau(\mathbf{s})$ of the form $(\mathbf{q}_0, \boldsymbol{\epsilon}), (\mathbf{q}_1, \mathbf{s}'[1]), \ldots, (\mathbf{q}_\tau, \mathbf{s}'[m])$, we denote the string $\mathbf{s}' = \mathbf{s}'[1]\mathbf{s}'[2]\ldots\mathbf{s}'[m]$ output by $\mathcal{N}$ along that execution by $[\![\rho_\tau]\!](\mathbf{s})$. We denote by $[\![\mathcal{N}_\tau]\!](\mathbf{s})$ the *set* of all possible perturbed outputs corresponding to the input string $\mathbf{s}$. Formally, $[\![\mathcal{N}_\tau]\!](\mathbf{s})$ is the set $\{\mathbf{s}' \mid \exists \rho_\tau(\mathbf{s})$ such that $\mathbf{s}' = [\![\rho_\tau]\!](\mathbf{s})\}$.

Robustness. Let $d : \Sigma^* \times \Sigma^* \to \mathbb{N}$ be a distance metric over a set Σ^* of strings. We can extend this metric to vectors of strings in the standard fashion. Let $\mathbf{w} = (w_1, \ldots, w_L)$, $\mathbf{v} = (v_1, \ldots, v_L)$ be two vectors of strings; then $d(\mathbf{w}, \mathbf{v}) = (d(w_1, v_1), \ldots, d(w_L, v_L))$.

Let τ_k denote the number of perturbations in the k^{th} internal channel in $\rho_\tau(\mathbf{s})$. Then, the *channel-wise perturbation count* in $\rho_\tau(\mathbf{s})$, denoted $\|\rho_\tau(\mathbf{s})\|$ is given by the vector $(\tau_1, \ldots, \tau_{|N|})$. We define robustness of a networked system as follows.

Definition 11.1.1 (Robust Networked System)**.** Given an upper bound $\boldsymbol{\delta} = \{\delta_1, \ldots, \delta_{|N|}\}$ on the number of possible perturbations in each internal channel, and an upper bound $\boldsymbol{\varepsilon} = (\varepsilon_1, \ldots, \varepsilon_{|O|})$ on the acceptable error in each external output channel of a networked system $\mathcal{N}$, we say that $\mathcal{N}$ is $(\boldsymbol{\delta}, \boldsymbol{\varepsilon})$-robust if:

$$\forall \mathbf{s} \in (\Sigma^{|I|})^\star, \forall \rho_\tau(\mathbf{s}) : \quad \|\rho_\tau(\mathbf{s})\| \leq \boldsymbol{\delta} \implies d(\, [\![\mathcal{N}]\!](\mathbf{s}), [\![\rho_\tau]\!](\mathbf{s})\,) \leq \boldsymbol{\varepsilon}.$$

11.2 Robustness Analysis

Checking if a networked system $\mathcal{N}$ is $(\boldsymbol{\delta}, \boldsymbol{\varepsilon})$-robust is equivalent to checking if, for each output channel $o_\ell \in O$ (with an error bound of ε_ℓ), $\mathcal{N}$ is $(\boldsymbol{\delta}, \varepsilon_\ell)$-robust. Thus, in what follows, we focus on the problem of checking robustness of the networked system $\mathcal{N}$ for a single output channel. Rephrasing the robustness definition from Def. 11.1.1, we need to check if for all input strings $\mathbf{s} \in (\Sigma^{|I|})^\star.(\#^{|I|})^\star$, and all runs $\rho_\tau(\mathbf{s})$ of $\mathcal{N}$, $\|\rho_\tau(\mathbf{s})\| \leq \boldsymbol{\delta}$ implies that $d([\![\mathcal{N}]\!]|_\ell(\mathbf{s}), [\![\rho_\tau]\!]|_\ell(\mathbf{s})) \leq \varepsilon_\ell$. Here, $[\![\mathcal{N}]\!]|_\ell(\mathbf{s})$, $[\![\rho_\tau]\!]|_\ell(\mathbf{s})$ respectively denote the projections of $[\![\mathcal{N}]\!](\mathbf{s})$ and $[\![\rho_\tau]\!](\mathbf{s})$ on the ℓ^{th} output channel. For simplicity in notation, henceforth, we drop the ℓ in the error bound on the channel, and denote it simply by ε.

Similar to the semantics of a networked system $\mathcal{N}$ with multiple output channels, we can define the semantics of $\mathcal{N}$ for the ℓ^{th} output channel using the tuple $(\Sigma^{|I|}, \Sigma, Q, \mathbf{q}_0, E|_\ell)$. Here, $E|_\ell$ denotes the projection of the transition relation E of $\mathcal{N}$ onto the ℓ^{th} output channel. To incorporate the addition of # symbols at the end of strings, the semantics of $\mathcal{N}$ is further modified to the tuple $(\Sigma^{|I|} \cup \{\#^{|I|}\}, \Sigma^\#, Q, \mathbf{q}_0, E^\#)$, where $E^\# = E|_\ell \cup \{(\mathbf{q}, ((\#, \ldots, \#), \#), \mathbf{q}) : \mathbf{q} \in Q\}$. Similarly, the tuple defining $\mathcal{N}_\tau$ is modified to $(\Sigma^{|I|} \cup \{\#^{|I|}{}_1, \ldots, \#^{|I|}{}_,\}\Sigma^\#, Q, \mathbf{q}_0, E^\#_\tau)$, where $E^\#_\tau$ includes $E^\#$ and all possible τ-transitions from each state.

In what follows, we define composite machines $\mathcal{A}$ that accept input strings certifying the non-robust behavior of a given networked system $\mathcal{N}$. In other words, $\mathcal{A}$ accepts a string $\mathbf{s} \in (\Sigma^{|I|})^\star.(\#^{|I|})^\star$ iff there exists a perturbed

execution $\rho_\tau(\mathbf{s})$ of $\mathcal{N}_\tau$ such that: $\|\rho_\tau(\mathbf{s})\| \leq \boldsymbol{\delta}$ and $d([\![\mathcal{N}]\!]|_\ell(\mathbf{s}), [\![\rho_\tau]\!]|_\ell(\mathbf{s})) > \varepsilon$. Thus, the networked system $\mathcal{N}$ is $(\boldsymbol{\delta}, \boldsymbol{\varepsilon})$-robust iff $\mathcal{L}(\mathcal{A})$ is empty.

11.2.1 Robustness Analysis for the Manhattan Distance Metric

The composite machine $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$ certifying non-robustness with respect to the Manhattan metric, is a nondeterministic, 1-reversal-bounded $(|N|+1)$-counter machine, i.e., in the class NCM($|N|+1$, 1). In each run on an input string $\mathbf{s}$, $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$ simultaneously does the following: (a) it simulates an unperturbed execution $\rho(\mathbf{s})$ of $\mathcal{N}$ and a perturbed execution $\rho_\tau(\mathbf{s})$ of $\mathcal{N}_\tau$, (b) keeps track of all the internal channel perturbations along $\rho_\tau(\mathbf{s})$, and (c) tracks the Manhattan distance between the outputs generated along $\rho(\mathbf{s})$ and $\rho_\tau(\mathbf{s})$.

Recall from Sec. 9.4, that the automaton $\mathcal{D}_M^{>\varepsilon}$, accepting pairs of strings with Manhattan distance greater than ε from each other, is in the class NCM(1,1). Let $\mathcal{D}_M^{>\varepsilon} = (\Sigma^{\#} \times \Sigma^{\#}, X_M, x_{0M}, Z_M, G_M, E_M, \{acc_M\})$, with all components of the tuple as defined in Sec. 9.4.

Formally, the machine $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$, in the class NCM($|N|+1$, 1) is defined as the tuple $(\Sigma^{|I|} \cup \{\#^{|I|}\}, X, \mathbf{x}_0, Z, G, E, F)$, where $X, \mathbf{x}_0, Z, G, E, F$ are respectively the set of states, initial state, set of counters, a finite set of integers, the transition relation and the final states of $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$. We define these below.

The set of states $X = Y \cup \{acc, rej\}$, where $Y \subseteq (Q \times Q \times X_M)$. Each state $\mathbf{x} \in Y$ of $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$ is a tuple $(\mathbf{q}, \mathbf{r}, x_M)$, where the component labeled $\mathbf{q}$ tracks the state of the unperturbed network $\mathcal{N}$, the component $\mathbf{r}$ tracks the state of the perturbed network $\mathcal{N}_\tau$, and x_M is a state in $\mathcal{D}_M^{>\varepsilon}$. The initial state $\mathbf{x}_0$ is

$(\mathbf{q}_0, \mathbf{q}_0, x_{0\,M})$. The set of counters $Z = \{z_1, \ldots, z_{|N|}\} \cup \{z\}$, where the counters $\{z_1, \ldots, z_{|N|}\}$ track the number of perturbations in each internal channel of $\mathcal{N}$, and the counter z tracks the Manhattan distance of the output strings. The initial value of each counter is 0. The set $G = \{0, -1, \delta_1, \delta_2, \ldots, \delta_{|N|}, \varepsilon\}$ is the set of all integers that can be used in tests on counter values, or by which any counter in Z can be incremented. The set F of final states is the singleton set $\{acc\}$.

The transition relation E of $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$ is constructed using the following steps:

1. *Initialization transition*:

   We add a single transition of the form:

$$\left((\mathbf{q}_0, \mathbf{q}_0, x_{0\,M}),\ \boldsymbol{\epsilon},\ \bigwedge_{k=1}^{|N|} z_k = 0 \wedge z = 0,\ (\mathbf{q}_0, \mathbf{q}_0, x_{0\,M}),\ (+\delta_1, \ldots, +\delta_{|N|},\ +\varepsilon) \right)$$

   In this transition, $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$ initializes each counter z_k for $k \in [1, |N|]$ to the error bound δ_k on the k^{th} internal channel, and also initializes the counter z to the error bound ε on the output channel. Note that no input symbol is consumed and there's no state change. Also, note that the counter test ensures that this transition can be taken only once from $\mathbf{x}_0$.

2. *Unperturbed network transitions*:

   For each pair of transitions in $E^{\#}$ and $E_\tau^{\#}$ from the same state, with

the same input symbol and output symbol, i.e., $(\mathbf{q}, \mathbf{a}, b, \mathbf{q}') \in E^{\#}$ and $(\mathbf{r}, \mathbf{a}, b, \mathbf{r}') \in E_{\tau}^{\#}$, and transitions of the form $(x_M, (b, b), z \geq 0, x_M, 0) \in E_M$, we add a transition of the following form to $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$:

$$\left((\mathbf{q}, \mathbf{r}, x_M), \quad \mathbf{a}, \quad \bigwedge_{k=1}^{|N|} z_k \geq 0 \wedge z \geq 0, \quad (\mathbf{q}', \mathbf{r}', x_M), \quad (0, \ldots, 0, 0) \right).$$

For each pair of transitions in $E^{\#}$ and $E_{\tau}^{\#}$ from the same state, with the same input symbol and different output symbols, i.e., $(\mathbf{q}, \mathbf{a}, b, \mathbf{q}') \in E^{\#}$ and $(\mathbf{r}, \mathbf{a}, b', \mathbf{r}') \in E_{\tau}^{\#}$, and transitions of the form $(x_M, (b, b'), z \geq 0, x_M, -1) \in E_M$, we add a transition of the following form to $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$:

$$\left((\mathbf{q}, \mathbf{r}, x_M), \quad \mathbf{a}, \quad \bigwedge_{k=1}^{|N|} z_k \geq 0 \wedge z \geq 0, \quad (\mathbf{q}', \mathbf{r}', x_M), \quad (0, \ldots, 0, -1) \right).$$

For each pair of transitions in $E^{\#}$ and $E_{\tau}^{\#}$ from the same state, with the same input symbol and different output symbols, i.e., $(\mathbf{q}, \mathbf{a}, b, \mathbf{q}') \in E^{\#}$ and $(\mathbf{r}, \mathbf{a}, b', \mathbf{r}') \in E_{\tau}^{\#}$, and transitions of the form $(x_M, (b, b'), z < 0, acc_M, 0) \in E_M$, we add transitions of the following form to $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$:

$$\left((\mathbf{q}, \mathbf{r}, x_M), \quad \mathbf{a}, \quad \bigwedge_{k=1}^{|N|} z_k \geq 0 \wedge z < 0, \quad (\mathbf{q}', \mathbf{r}', acc_M), \quad (0, \ldots, 0, 0) \right).$$

In each of the above transitions, $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$ consumes an input symbol $\mathbf{a}$ and simulates a pair of unperturbed transitions on $\mathbf{a}$ in the first two components of its state. The distance between the corresponding outputs of $\mathcal{N}$

(b and b' above) is tracked by the counter z and the third component of the state of $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$. Note that the values of all counters z_k, for $k \in [1, |N|]$ are required to be non-negative in the source state and remain unmodified in each transition.

3. *Perturbed network transitions*:

   From each state $\mathbf{x} \in Y$, we add transitions of the form:

$$\left((\mathbf{q}, \mathbf{r}, x_M), \quad \boldsymbol{\epsilon}, \quad \bigwedge_{k=1}^{|N|} z_k \geq 0, \quad (\mathbf{q}, \mathbf{r}_\tau, x_M), \quad \mathbf{g} \right)$$

   In each such transition, $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$ simulates a τ-transition $(\mathbf{r}, \boldsymbol{\epsilon}, \boldsymbol{\epsilon}, \mathbf{r}_\tau) \in E_\tau^{\#}$. In the transition, $\mathbf{g}$ denotes a $(|N|+1)$-length vector, with entries in $\{0, -1\}$: for $k \in [1, |N|]$, $g_k = -1$ iff the k^{th} internal channel is perturbed in $(\mathbf{r}, \boldsymbol{\epsilon}, \boldsymbol{\epsilon}, \mathbf{r}_\tau)$, and $g_{|N|+1} = 0$. Thus, we model a perturbation on the k^{th} internal channel by decrementing the (non-negative) z_k counter of $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$. Note that in these transitions, no input symbol is consumed, and the first and third components, i.e. $\mathbf{q}$ and x_M remain unchanged.

4. *Rejecting transitions*:

   From each state $\mathbf{x} \in Y$, we add transitions of the form:

$$\left((\mathbf{q}, \mathbf{r}, x_M), \quad \boldsymbol{\epsilon}, \quad \bigvee_{k=1}^{|N|} z_k < 0, \quad \mathit{rej}, \quad \mathbf{0} \right)$$

From the state *rej*, for all $\mathbf{a}$, we add a transition: $(rej, \mathbf{a}, true, rej, \mathbf{0})$.

Thus, we add a transition to a designated rejecting state whenever the value of some counter z_k, for $k \in [1, |N|]$ goes below 0, i.e., whenever the perturbation count in some k^{th} internal channel exceeds the error bound $\boldsymbol{\delta}_k$. Once in the state *rej*, $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$ ignores any further input read, and remains in that state.

5. *Accepting transitions*:

   Finally, from each state $(\mathbf{q}, \mathbf{r}, acc_M) \in Y$, we add transitions of the form:

$$\left((\mathbf{q}, \mathbf{r}, x_M), \quad \boldsymbol{\epsilon}, \quad \bigwedge_{k=1}^{|N|} z_k \geq 0, \quad acc, \quad \mathbf{0} \right)$$

   We add a transition to the unique accepting state whenever $x_M = acc_M$ and $\bigwedge_{k=1}^{|N|} z_k \geq 0$. The first criterion ensures that $d_M([\![\mathcal{N}]\!]|_\ell(\mathbf{s}), [\![\rho_\tau]\!]|_\ell(\mathbf{s})) > \varepsilon$ (as indicated by reaching the accepting state in $\mathcal{D}_M^{>\varepsilon}$). The second criterion ensures that $\|\rho_\tau(\mathbf{s})\| \leq \boldsymbol{\delta}$, i.e., the run $\rho_\tau(\mathbf{s})$ of $\mathcal{N}$ on $\mathbf{s}$ models perturbations on the network that respect the internal channel error bounds.

Theorem 11.2.1. *Given an upper bound $\boldsymbol{\delta}$ on the number of perturbations in the internal channels, and an upper bound ε on the acceptable error for a particular output channel, the problem of checking if the networked system $\mathcal{N}$ is $(\boldsymbol{\delta}, \varepsilon)$-robust with respect to the Manhattan distance can be accomplished in*

NLOGSPACE *in the number* $|Q|$ *of network states, the number* $|E_\tau^{\#}|$ *of perturbed network transitions,* $\boldsymbol{\delta}$ *and* ε*.*

Proof. We note that the construction of $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$ reduces the problem of checking $(\boldsymbol{\delta},\varepsilon)$-robustness for $\mathcal{N}$ (w.r.t. the Manhattan distance) to checking emptiness of $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$. From the construction of $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$, we further note that the size of $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$ is polynomial in the number $|Q|$ of network states, the number $|E_\tau^{\#}|$ of the set of perturbed network transitions (which dominates the size $|E|$ of the set of unperturbed network transitions) and in $\boldsymbol{\delta}$ and ε. The theorem then follows from Lem. 9.3.1, which states that emptiness checking of $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$ can be solved in NLOGSPACE in its size. □

11.2.2 Robustness Analysis for the Levenshtein Distance Metric

The composite machine $\mathcal{A}_L^{\boldsymbol{\delta},\varepsilon}$, certifying non-robustness with respect to the Levenshtein distance metric, is a nondeterministic 1-reversal-bounded $|N|$-counter machine, i.e., in the class NCM($|N|$,1). Similar to $\mathcal{A}_M^{\boldsymbol{\delta},\varepsilon}$, in each run on an input string $\mathbf{s}$, $\mathcal{A}_L^{\boldsymbol{\delta},\varepsilon}$ simultaneously does the following: (a) it simulates an unperturbed execution $\rho(\mathbf{s})$ of $\mathcal{N}$ and a perturbed execution $\rho_\tau(\mathbf{s})$ of $\mathcal{N}_\tau$, (b) keeps track of all the internal channel perturbations along $\rho_\tau(\mathbf{s})$, and (c) tracks the Levenshtein distance between the outputs generated along $\rho(\mathbf{s})$ and $\rho_\tau(\mathbf{s})$.

Recall from Sec. 9.5, that the automaton $\mathcal{D}_L^{>\varepsilon}$, accepting pairs of strings with Levenshtein distance greater than ε from each other, is a DFA. Let $\mathcal{D}_L^{>\varepsilon}$ $= ((\Sigma^{\#} \times \Sigma^{\#}), Q_L, \mathbf{q}_{0\,L}, \Delta_L, \{acc_L\})$.

Formally, $\mathcal{A}_L^{\boldsymbol{\delta},\varepsilon}$, in the class NCM($|N|$,1), is defined as the tuple $(\Sigma^{|I|} \cup \{\#^{|I|}\}, X, \mathbf{x}_0, Z, G, E, F)$, where $X, \mathbf{x}_0, Z, G, E, F$ are respectively the set of states, initial state, set of counters, a finite set of integers, the transition relation and the final states of $\mathcal{A}_L^{\boldsymbol{\delta},\varepsilon}$. We define these below.

The set of states $X = Y \cup \{acc, rej\}$, where $Y \subseteq (Q \times Q \times Q_{Lev})$. The initial state of $\mathcal{A}_L^{\boldsymbol{\delta},\varepsilon}$, $\mathbf{x}_0$, is given by the tuple $(\mathbf{q}_0, \mathbf{q}_0, \mathbf{q}_{0\,L})$. The set of counters $Z = \{z_1, \ldots, z_{|N|}\}$ tracks the number of perturbations in each internal channel of $\mathcal{N}$, and the initial value of each counter is 0. The set $G = \{0, -1, \delta_1, \delta_2, \ldots, \delta_{|N|}\}$, and the set of final states is the singleton set $\{acc\}$.

The transition relation E of $\mathcal{A}_L^{\boldsymbol{\delta},\varepsilon}$ is constructed using the following steps:

1. *Initialization transition*:

   From the initial state $\mathbf{x}_0$, we add a single transition of the form:

$$\left((\mathbf{q}_0, \mathbf{q}_0, \mathbf{q}_{0\,L}), \quad \boldsymbol{\epsilon}, \quad \bigwedge_{k=1}^{|N|} z_k = 0, \quad (\mathbf{q}_0, \mathbf{q}_0, \mathbf{q}_{0\,L}), \quad (+\delta_1, \ldots, +\delta_{|N|}) \right)$$

   In this transition, $\mathcal{A}_L^{\boldsymbol{\delta},\varepsilon}$ sets each counter z_k to the error bound δ_k on the k^{th} internal channel, without consuming an input symbol or changing state. The counter test ensures that this transition can be taken only once from $\mathbf{x}_0$.

2. *Unperturbed network transitions*:

   For each pair of transitions in $E^{\#}$ and $E^{\#}_{\tau}$ on the same input symbol from the same state, i.e., $(\mathbf{q}, \mathbf{a}, b, \mathbf{q}') \in E^{\#}$ and $(\mathbf{r}, \mathbf{a}, b', \mathbf{r}') \in E^{\#}_{\tau}$, and transitions of the form $(\mathbf{q}_L, (b, b'), \mathbf{q}'_L) \in \Delta_L$, we add a transition of the following form to $\mathcal{A}^{\boldsymbol{\delta},\varepsilon}_L$:

$$\left((\mathbf{q}, \mathbf{r}, \mathbf{q}_L), \quad \mathbf{a}, \quad \bigwedge_{k=1}^{|N|} z_k \geq 0, \quad (\mathbf{q}', \mathbf{r}', \mathbf{q}'_L), \quad \mathbf{0} \right)$$

   In each such transition, $\mathcal{A}^{\boldsymbol{\delta},\varepsilon}_L$ consumes an input symbol $\mathbf{a} \in \Sigma^{|I|} \cup \{\#^{|I|}\}$ and simulates a pair of unperturbed transitions on $\mathbf{a}$ in the first two components of its state. The distance between the corresponding outputs of $\mathcal{N}$ (b and b' above) is tracked by the third component. Note that in such transitions, all counter values are required to be non-negative in the source state and are not modified.

3. *Perturbed network transitions*:

   From each state $\mathbf{x} \in Y$, we add transitions of the form:

$$\left((\mathbf{q}, \mathbf{r}, \mathbf{q}_L), \quad \boldsymbol{\epsilon}, \quad \bigwedge_{k=1}^{|N|} z_k \geq 0, \quad (\mathbf{q}, \mathbf{r}_\tau, \mathbf{q}_L), \quad \mathbf{g} \right)$$

   In each such transition, $\mathcal{A}^{\boldsymbol{\delta},\varepsilon}_L$ simulates a τ-transition $(\mathbf{r}, \boldsymbol{\epsilon}, \boldsymbol{\epsilon}, \mathbf{r}_\tau) \in E^{\#}_{\tau}$. In the transition, $\mathbf{g}$ denotes a $|N|$-length vector with entries in $\{0, -1\}$, where $g_k = -1$ iff the k^{th} internal channel is perturbed in $(\mathbf{r}, \boldsymbol{\epsilon}, \boldsymbol{\epsilon}, \mathbf{r}_\tau)$.

Note that in these transitions, no input symbol is consumed, and the first and third components, i.e. $\mathbf{q}$ and $\mathbf{q}_L$ remain unchanged.

4. *Rejecting transitions*:

   From each state $\mathbf{x} \in Y$, we add transitions of the form:

$$\left((\mathbf{q}, \mathbf{r}, \mathbf{q}_L), \quad \boldsymbol{\epsilon}, \quad \bigvee_{k=1}^{|N|} z_k < 0, \quad rej, \quad \mathbf{0} \right)$$

   From the state rej, for all $\mathbf{a}$, we add a transition: $(rej, \mathbf{a}, true, rej, \mathbf{0})$.

5. *Accepting transitions*:

   Finally, from each state $(\mathbf{q}, \mathbf{r}, acc_L) \in Y$, we add transitions of the form:

$$\left((\mathbf{q}, \mathbf{r}, \mathbf{q}_L), \quad \boldsymbol{\epsilon}, \quad \bigwedge_{k=1}^{|N|} z_k \geq 0, \quad acc, \quad \mathbf{0} \right)$$

   We add a transition to the unique accepting state whenever $\mathbf{q}_L = acc_L$ and $\bigwedge_{k=1}^{|N|} z_k \geq 0$. The first criterion ensures that $d_L([\![\mathcal{N}]\!]|_\ell(\mathbf{s}), [\![\rho_\tau]\!]|_\ell(\mathbf{s})) > \varepsilon$ (as indicated by reaching the accepting state in $\mathcal{D}_L^{>\varepsilon}$). The second criterion ensures that $\|\rho_\tau(\mathbf{s})\| \leq \boldsymbol{\delta}$, i.e., the run $\rho_\tau(\mathbf{s})$ of $\mathcal{N}$ on $\mathbf{s}$ models perturbations on the network that respect the internal channel error bounds.

Theorem 11.2.2. *Given an upper bound $\boldsymbol{\delta}$ on the number of perturbations in the internal channels, and an upper bound ε on the acceptable error for a particular output channel, the problem of checking if the networked system $\mathcal{N}$ is $(\boldsymbol{\delta}, \varepsilon)$-robust with respect to the Levenshtein distance can be accomplished in* PSPACE *in ε.*

Proof. We first note that the construction of $\mathcal{A}_L^{\boldsymbol{\delta},\varepsilon}$ reduces the problem of checking $(\boldsymbol{\delta}, \varepsilon)$-robustness of $\mathcal{N}$ (w.r.t. the Levenshtein distance) to checking emptiness of $\mathcal{A}_L^{\boldsymbol{\delta},\varepsilon}$. From the construction of $\mathcal{A}_L^{\boldsymbol{\delta},\varepsilon}$ and $\mathcal{D}_L^{>\varepsilon}$, we further note that $\mathcal{A}_L^{\boldsymbol{\delta},\varepsilon}$ belongs to the class NCM($|N|$, 1) and its size is $\mathcal{O}((\varepsilon|\Sigma|)^{4\varepsilon})$, and is polynomial in the number $|Q|$ of network states, the number $|E_\tau^{\#}|$ of the set of perturbed network transitions (which dominates the size $|E|$ of the set of unperturbed network transitions) and in $\boldsymbol{\delta}$. The theorem then follows from Lem. 9.3.1, which states that emptiness checking of $\mathcal{A}_L^{\boldsymbol{\delta},\varepsilon}$ can be solved in NLOGSPACE in its size. □

Chapter 12

Bibliographic Notes

Many tasks in computing involve the evaluation of functions from strings to strings. Such functions are often naturally represented as finite-state string transducers [19, 62, 93, 125]. For example, inside every compiler is a transducer that maps user-written text to a string over tokens, and authors of web applications routinely write transducers to sanitize user input. Systems for natural language processing use transducers for executing morphological rules, correcting spelling, and processing speech. Many of the string algorithms at the heart of computational biology or image processing are essentially functional transducers. The transducer representation of functions has been studied thoroughly over the decades, and many decision procedures and expressiveness results about them are known [93, 125]. However, the behavior of finite-state transducers under uncertain operating environments has been less well-studied.

Recently, there has been a growing interest in the study of robustness in the formal methods and software engineering communities. The initial papers by Majumdar and Saha [88] and by Chaudhuri et al [22–24] study continuity and robustness analysis of infinite-state programs. While these papers reason

about programs that manipulate numbers, we focus on robustness analysis of programs manipulating strings [108]. As the underlying metric topologies are quite different, these approaches are essentially complementary to ours. It is also important to note that the analyses presented in these papers is incomplete and their scope does not extend to networked systems with channel errors like ours [107].

More recent papers have aimed to develop a notion of robustness for reactive systems. In [119], the authors present polynomial-time algorithms for the analysis and synthesis of robust transducers. Their notion of robustness is one of input-output stability, that bounds both the deviation of the output from disturbance-free behaviour under bounded disturbance, as well as the persistence of the effect on the output of a sporadic disturbance. Unlike our distance metrics which quantify the cost of transforming one string into another, their distances are measured using cost functions that map *each* string to a nonnegative integer.

In [16, 89, 123], the authors develop different notions of robustness for reactive systems, with ω-regular specifications, interacting with uncertain environments. In [89], the authors present metric automata, which are automata equipped with a metric on states. The authors assume that at any step, the environment can perturb any state q to a state at most $\gamma(q)$ distance away, where γ is some function mapping states to real numbers. A winning strategy for a finite-state or Büchi automaton $\mathcal{A}$ is a strategy that satisfies the corresponding acceptance condition (stated as reachability of states in F or as

infinitely often visiting states in F respectively). Such a winning strategy is defined to be σ-robust if it is a winning strategy for $\mathcal{A}$ where the set F' characterizing the acceptance condition includes all states at most $\sigma.\sup_{q \in F}\gamma(q)$ distance away from the F. We note that while there are some similarities in how a disturbance is modeled, our approach is quite different, as we quantify and analyze the effect of errors over time, and do not associate metrics with individual states.

In [40], the authors study robustness of sequential circuits w.r.t. a *common suffix distance* metric. Their notion of robustness essentially bounds the persistence of the effect of a sporadic disturbance in the input of a sequential circuit. To be precise, a circuit is said to be robust iff the position of the last mismatch in any pair of output sequences is a bounded number of positions from the last mismatch position in the corresponding pair of input sequences. The authors present a polynomial-time algorithm to decide robustness of sequential circuits modeled as (single) Mealy machines. The metric and its subsequent treatment developed in this paper is useful for analyzing circuits; however, for networked systems communicating via strings, metrics such as the Manhattan and Levenshtein distances provide a more standard way to measure the effect of errors.

In [43], the authors present modeling techniques for cyber-physical systems. Further, the authors also discuss the challenges of including a network in a cyber-physical system. A key observation is that to maintain discrete-event semantics of components in such a system, it is important to have a common

sense of time across all components. A critical requirement in such systems is that the communication remain synchronized, which is typically fulfilled by using protocols that bound the allowed drift in the value of the global clock. In our model, we do not analyze such details, and abstract them away, assuming that some underlying protocol ensures synchronous communication. Recent papers on wireless control networks [98, 99], classic models like Kahn process networks [80], and languages like Esterel [18] have made similar assumptions about synchronous communication.

Work in the area of robust control seeks to analyze and design networked control systems where communication between sensors, the controller, and actuators occurs over unreliable networks such as wireless networks [1]. On the other hand, work on wireless control networks [98, 99] focuses on design of distributed controllers where the components of the controller communicate over unreliable wireless networks. In such applications, robustness typically means desirable properties of the control loop such as stability. We note that these papers typically assume a synchronous communication schedule as supported by wireless industrial control protocols such as ISA 100 and WirelessHART.

Part V

Conclusions

Chapter 13

Conclusions

Software systems today are increasingly complex, ubiquitous and often interact with each other or with the physical world. While their reliability has never been more critical, these systems remain vulnerable to the fallibility of human programmers as well as the unpredictability of their operating environments. The only solution that holds promise is increased usage of ***meta-programs*** *to help analyze, debug and synthesize programs, given a precise characterization of reliable program behaviour.*

In this dissertation, we have presented several ***meta-programs***, i.e., algorithms, to help analyze, debug and synthesize various models of programs. In particular, we have developed algorithms for debugging sequential programs, for synthesizing synchronization for concurrent programs and for verifying the robustness of certain systems modeled using transducers. In this chapter, we conclude this dissertation by summarizing our contributions in each of these domains, and outlining avenues for future work.

13.1 Summary of Contributions

13.1.1 Debugging of Sequential Programs

A broad and informal statement of the (automated) program debugging problem is to compute a correct program $\widehat{\mathcal{P}}$ that is obtained by *suitably* modifying an erroneous program $\mathcal{P}$. This problem is undecidable in general; it is hard to formalize; moreover, it is particularly challenging to assimilate and mechanize the customized, expert human intuition involved in the choices made in manual program debugging. Our contributions in this domain are as follows.

1. **Methodical Problem Formulation:**

   We present several problem definitions that help formalize the program debugging problem and enable automation.

   (a) *Update Schemas*: We propose a problem formulation in which, along with an erroneous program $\mathcal{P}$, we are given a set $\mathcal{U}$ of update schemas, describing a class of permissible modifications of a statement in $\mathcal{P}$. The goal is to compute a correct program $\widehat{\mathcal{P}}$ that is obtained from $\mathcal{P}$ by applying suitable update schemas from $\mathcal{U}$ to the statements in $\mathcal{P}$. While a typical debugging routine begins with fault localization and is followed by error elimination, our update schema-based formulation obviates the need for a separate fault localization phase, and enables us to directly focus on error elimination or program repair.

(b) *Cost-awareness*: We further propose a problem formulation in which, we are also given a repair budget and a cost function that charges each application of an update schema to a program statement some user-defined cost. The goal is now to compute a repaired program $\widehat{\mathcal{P}}$ whose total modification cost does not exceed the repair budget.

(c) *Template-based Repair*: Finally, we propose a template-based problem formulation, in which the additional goal is to compute $\widehat{\mathcal{P}}$ such that the syntax of any repaired expression matches some user-specified expression template.

All of the above problem formulations provide ways to incorporate expert programmer intuition and intent in automated program debugging. Insightful choices for update schemas, cost functions and templates can help prune the search space for repaired programs, and help compute a repaired program *similar* to what the programmer may have in mind.

2. **Predicate abstraction-based Solution Framework:**

We present a predicate abstraction-based solution framework for the above problems that can repair infinite-state, imperative, sequential programs. As part of this framework, we make the following contributions.

(a) We present a sound and complete algorithm for automatic repair of Boolean programs that meet some syntactic requirements and can be repaired by a single application of an update schema. Our

approach targets total correctness with respect to a specification in the form of a precondition, postcondition pair.

(b) We present a sound and complete algorithm for automatic repair of arbitrary Boolean programs (or, pushdown systems), annotated with multiple assertions, with respect to partial correctness. This algorithm can repair Boolean programs by modifying them in multiple program locations using suitable update schemas. Along with a repaired program, the algorithm also generates a proof of correctness composed of inductive assertions.

(c) We present techniques to concretize a repaired Boolean program, with and without user-supplied expression templates, to obtain a concrete repaired program $\widehat{\mathcal{P}}$.

(d) We present experimental results for repairing C programs, using a prototype implementation based on SMT solving.

13.1.2 Synchronization Synthesis for Concurrent Programs

Extant work in this domain has focused on either propositional temporal logic specifications with simplistic models of concurrent programs, or more refined program models with the specifications limited to just safety properties. Moreover, there has been limited effort in developing adaptable and fully-automatic synthesis frameworks that are capable of generating synchronization at different levels of abstraction and granularity. Our contributions in this domain are as follows.

1. **Synthesis of Low-level Synchronization:**

   (a) We present a framework that takes unsynchronized sequential process skeletons along with a propositional temporal logic specification of their global concurrent behaviour, and automatically generates a concurrent program with synchronization code ensuring correct global behaviour. The synthesized synchronization code can be coarse-grained or fine-grained, and is based on readily-implementable synchronization primitives such as locks and condition variables. The overall method is fully automatic, sound and complete.

   (b) As part of the framework, we present algorithms to compile high-level synchronization actions in the form of guarded commands into coarse-grained and fine-grained synchronization code based on locks and condition variables. The ability to automatically synthesize fine-grained synchronization code is noteworthy; programmers often restrict themselves to using coarse-grained synchronization for its inherent simplicity. In fact, manual implementations of synchronization code using wait/notify operations on condition variables are particularly hard to get right in the presence of multiple locks.

   (c) We provide detailed proofs of the correctness of the compilations with respect to a useful subset of propositional CTL.

   (d) We use our prototype tool to successfully synthesize synchronization code for concurrent Java programs such as an airport ground traffic

simulator program, readers-writers and dining philosophers.

2. **Generalized Synchronization Synthesis:**

   (a) We propose a generalized framework that can synthesize synchronization for real-world shared-memory concurrent programs, given unsynchronized processes, and temporal logic properties over both control and data variables.

   (b) For the specification language, we propose an extension of propositional CTL that facilitates expression of both safety and liveness properties over control and data variables.

   (c) We present an extension of the synthesis procedure of [47] for our proposed specification language and program model. This extension enables synchronization synthesis for finite-state concurrent programs composed of processes that may have local and shared variables, may be straight-line or branching programs, may be ongoing or terminating, and may have program-initialized or user-initialized variables.

   (d) We further present compilations of high-level synchronization actions into lower-level coarse-grained or fine-grained synchronization based on locks and condition variables for our proposed class of programs and specifications.

13.1.3 Robustness Analysis of Discrete Systems

On the one hand, techniques and results from relevant, mature areas such as robust control are not directly applicable to robustness analysis of systems with large amounts of discretized, discontinuous behavior. On the other hand, traditional program verification techniques do not provide a quantitative measure of the sensitivity of system behaviour to uncertainty in the operating environment. Hence, robustness analysis of software programs used in heterogeneous settings necessitates development of new theoretical frameworks and algorithms. Our contributions in this domain are as follows.

1. **Methodical Problem Formulation:** We develop notions of robustness for certain systems modeled using transducers with respect to specific sources of uncertainty.

   (a) We formally define a notion of robustness for functional string transducers in the presence of input perturbations.

   (b) We present a formal model of a synchronous networked system of processes (Mealy machines), and define a notion of robustness for computations of such networked systems when the communication channels are prone to perturbation.

2. **Distance-tracking Automata Constructions:**
   We provide constructions for automata that can track various distance metrics between two strings.

(a) We define reversal-bounded counter machines that can track the the traditional and generalized versions of the Manhattan distance between strings.

(b) We define deterministic finite automata that can track the the traditional and generalized versions of the Levenshtein distance between strings.

3. **Automata-theoretic Framework for Robustness Analysis:**
We present automata-theoretic decision procedures that utilize distance-tracking automata to reduce the problem of robustness verification of our systems to the problem of checking the emptiness of certain carefully constructed automata.

(a) For robustness verification of Mealy machines and functional transducers, we define product machines, composed of *input automata*, *pair-transducers* and *output automata*, that essentially accept all input string pairs that certify the non-robustness of the transducer under consideration. For Mealy machines, the automata constructions are fairly straight-forward, and the decision procedures for checking robustness with respect to the generalized Manhattan and Levenshtein distances are in NLogspace and PSpace, respectively. For functional transducers, the output automata constructions are more involved, and the decision procedures for checking robustness with

respect to the generalized Manhattan and Levenshtein distances are in PSPACE and EXPSPACE, respectively.

(b) For robustness verification of synchronous networked systems of transducers, we define composite machines that accept input string pairs certifying the non-robust behavior of the networked system under consideration. Our decision procedures for checking robustness with respect to the Manhattan and Levenshtein distances are in NLOGSPACE and PSPACE, respectively.

13.2 Future Work

While we have presented several algorithms in this dissertation to help analyze, debug and synthesize various program models with respect to different characterizations of reliable program behaviour, our work is by no means complete. In what follows, we discuss extensions that can enhance the scope and performance of our current body of work. When applicable, we also outline avenues for future work that go beyond the specific methodologies of this dissertation, but share the same goals.

13.2.1 Debugging of Sequential Programs

It may be possible to extend the approach presented in Chapter 3 to repair Boolean programs using simultaneous applications of multiple update schemas. The basic idea would be to formulate a suitable QBF over variables representing unknown/suspect expressions in the incorrect Boolean program, and

extract the repaired expressions from the certificate of validity of the QBF using a QBF solver. However, we believe, the approach presented in Chapter 4 has more potential as an automated program debugging methodology. We describe some possible extensions below:

1. Our current tool can be improved in many ways. We can generalize the tool to permit different kinds of cost functions, statement deletions, handling of procedures and fully-automatic concretization. Moreover, we believe the performance of our current implementation can be improved by experimenting with different SMT-encodings of the cost-aware repairability conditions.

2. As mentioned in Chapter 4, our algorithm can be extended to handle total correctness by additionally computing ranking functions and using the method of well-founded sets along with the method of inductive assertions.

3. While the algorithm presented in Chapter 4 separates the computation of a repaired Boolean program $\widehat{\mathcal{B}}$ from its concretization to obtain $\widehat{\mathcal{P}}$, this separation is not necessary. In fact, the separation may be sub-optimal — it may not possible to concretize all modified statements of a computed $\widehat{\mathcal{B}}$, while there may exist some *other* concretizable $\widehat{\mathcal{B}}$. The solution is to directly search for $\widehat{\mathcal{B}}$ such that all modified statements of $\widehat{\mathcal{B}}$ are concretizable. This can be done by combining the constraints presented in Sec. 4.3 with the one in (4.4). In particular, the set $Unknown$ in (4.4)

can be modified to include unknown expressions/template parameters needed in the formulas in Sec. 4.3, and $CRC_{PC}(\pi)$ can be modified to include the inner quantifier-free constraints in the formulas in Sec. 4.3.

4. In our algorithm, we set $\mathcal{I}_{entry_0}$ to `true` and require the other inductive assertions to simply ensure the validity of all the $CRC_{PC}(\pi)$ conditions. Note however, that since both the program $\widehat{\mathcal{B}}$ and its set $\mathcal{I}_\Lambda$ of inductive assertions are unknown, it is possible to pick $\widehat{\mathcal{B}}$ and $\mathcal{I}_\Lambda$ such that the inductive assertions are needlessly restrictive and $\widehat{\mathcal{B}}$ has only a few feasible execution paths. The solution to this problem is to compute the weakest possible set of inductive assertions and a least restrictive $\widehat{\mathcal{B}}$. The first part may be accomplished by iteratively weakening the inductive assertions inferred from (4.4), and the second part may be similarly accomplished by iteratively weakening the expression modifications inferred from (4.4).

5. Our framework can be used to define and handle more interesting update schemas. In particular, one may choose to permit *insertion* of statements to significantly expand the search space for repaired programs. It may be possible to adapt program synthesis techniques (such as [82]) to compute localized repairs consisting of inserted program fragments. Besides statement modification, it should also be possible to use our framework to infer modified program preconditions by keeping $\mathcal{I}_{entry_0}$ unknown as well.

6. In our experiments, we found that the non-determinism inherent to Boolean programs can interfere with the scalability of the technique. Hence, it is worth investigating the efficiency and effectiveness of applying the method of inductive assertions to repair concrete programs directly. We emphasize that the method of inductive assertions for arbitrary concrete programs would be incomplete in general and might have to rely on user-supplied templates for all unknown inductive assertions and expressions, thereby necessitating more user involvement.

We note that Boolean programs can model both sequential and combinational circuits, and hence, our techniques can be used for repairing such circuits.

13.2.2 Synchronization Synthesis for Concurrent Programs

We wish to extend our current framework for efficient synthesis of synchronization for large finite-state or infinite-state concurrent programs. In particular, we wish to focus on defining and computing sound, finitary abstractions of the tableaux corresponding to large finite-state or infinite-state concurrent programs. To this end, it might be helpful to investigate the adaption of automata-theoretic approaches for deciding the satisfiability of various *constraint temporal logics* [31, 54] as well as predicate abstraction-based tableaux construction.

Another approach for efficient synthesis of synchronization is to avoid constructing and exploring the entire product graph or tableaux corresponding to the concurrent program, and instead use clever techniques to decompose the

task at hand. One such technique is to explore the set of feasible permutations of the statements in a given execution trace of the concurrent program in one step [124], before moving on to a new execution trace with a different set of statements. One of the challenges here is to ensure that the synchronization synthesized in different steps can be composed with each other, and that the overall synchronization is as *permissive* as possible.

Finally, we note that a related goal is repair of existing synchronization code in incorrect concurrent programs. This is an important problem in legacy code as it may not always be easy to remove existing synchronization code [79] before synthesizing new synchronization code.

13.2.3 Robustness Analysis of Discrete Systems

There are a few directions in which our robustness analysis frameworks can be developed further. The first is a more extensive treatment of distance metrics. It is clear that that the right distance metric to use to measure perturbation depends on the task at hand. For instance, for natural language or image processing or even compiler applications, the Levenshtein or Manhattan distance metrics are likely to suffice. For sequential circuits, the authors in [40] argue that the common suffix distance metric is more suitable. For cyberphysical systems, where the symbol sequences could represent a wide range of digital signals, one must track the *magnitude* of the signals. This necessitates defining and computing distances that are based on mapping individual symbols or symbol sequences to numbers [119]. To summarize, it would be interesting

to identify specific applications and related distance metrics, and extend our robustness analysis framework accordingly.

The second direction is a generalization of the error model and subsequently, the robustness definition. In our work, we only focus on internal channel errors in a network or, on input perturbations. However, in a real-world scenario, there can be multiple *simultaneous* sources of uncertainty such as sensor and actuator noise, modeling errors and process failures. A comprehensive robustness analysis should thus check if small disturbances in the inputs or internal channels or processes result in small deviations in the system behaviour.

We also wish to investigate robustness analysis of other program models such as traditional software sequential and concurrent programs, and perhaps synthesis of robust programs.

Appendix

Appendix 1

Select Proofs

In this appendix, we present the proofs of Lem. 6.3.1 and Lem. 6.3.2. We begin by introducing some preliminary definitions (cf. [45, 92, 95]).

1.1 Basic Definitions

Definition 1.1.1. (Stuttering Equivalent Paths)
Let $\mathcal{M}_u = (S_u, R_u, L_u)$ and $\mathcal{M}_v = (S_v, R_v, L_v)$ be two structures/models over the same set of atomic propositions AP, and let $B \subseteq S_u \times S_v$ be a relation. Paths $\pi_u = u_0, u_1, \ldots$ and $\pi_v = v_0, v_1, \ldots$ are called stuttering $B-$equivalent iff there exist infinite sequences of natural numbers $i_0 = 0 < i_i < i_2 < \ldots$ and $k_0 = 0 < k_1 < k_2$ such that for all $j \geq 0$, the following condition is true:
$\forall q, r \in \mathbb{N} : \ (i_j \leq q \leq i_{j+1}) \wedge (k_j \leq r \leq k_{j+1}) \Rightarrow (u_q, v_r) \in B$.
If B is defined such $\forall u \in S_u$, $v \in S_v$: $(u, v) \in B$ iff $L_u(u) = L_v(v)$, then the paths π_u and π_v are simply called stuttering equivalent.

Definition 1.1.2. (Stuttering Path Equivalence)
Structures $\mathcal{M}_u = (S_u, R_u, L_u)$ and $\mathcal{M}_v = (S_v, R_v, L_v)$, with the same set of atomic propositions AP, are called stuttering path equivalent (denoted $\mathcal{M}_u \equiv_{SPE} \mathcal{M}_v$) iff

1. for every path π_u starting from an initial state $u_0 \in S_u$, there exists a stuttering equivalent path π_v starting from an intial state $v_0 \in S_v$, and

2. a symmetric condition holds with the roles of u_0 and v_0 reversed.

Lemma 1.1.1. *Given structures $\mathcal{M}_u, \mathcal{M}_v$ with $\mathcal{M}_u \equiv_{SPE} \mathcal{M}_v$, and a $\mathsf{CTL}^* \setminus \mathsf{X}$ formula ϕ of the form $\mathsf{A}\psi$ or $\mathsf{E}\psi$, where ψ is in $\mathsf{LTL} \setminus \mathsf{X}$, $\mathcal{M}_u \models \phi$ iff $\mathcal{M}_v \models \phi$.*

Definition 1.1.3. (Stuttering Simulation)
Given a structure $\mathcal{M} = (S, R, L)$, a relation $B \subseteq S \times S$ is a stuttering simulation iff for any $(u, w) \in B$:

1. $L(u) = L(w)$,

2. for every path starting from u, there exists a stuttering B-equivalent path starting from w in $\mathcal{M}$.

Definition 1.1.4. (Well-founded Simulation)
Given a Kripke structure $\mathcal{M} = (S, R, L)$, a relation $B \subseteq S \times S$ is a well-founded simulation iff for any $(u, w) \in B$

1. $L(u) = L(w)$, and,

2. there exist functions $rank_1 : S \times S \times S \mapsto \mathbb{N}$, and $rank_2 : S \times S \mapsto W$, with $\langle W, \prec \rangle$ well-founded, such that $\forall t : (u, t) \in R$

   (a) $\exists v : ((w, v) \in R \wedge (t, v) \in B) \quad \vee$

   (b) $(t, w) \in B \wedge rank_2(t, w) \prec rank_2(u, w) \quad \vee$

(c) $\exists v : ((w,v) \in R \wedge (u,v) \in B \wedge rank_1(v,u,t) \prec rank_1(w,u,t))$.

Lemma 1.1.2. *Given Kripke structure $\mathcal{M} = (S, R, L)$, a relation $B \subseteq S \times S$ is a well-founded simulation iff B is a stuttering simulation.*

Lemma 1.1.3. *Given Kripke structure $\mathcal{M} = (S, R, L)$, a stuttering simulation B, and a formula ϕ in* ACTL$^*\backslash X$*, if $(u, w) \in B$ and $\mathcal{M}, w \models \phi$, then $\mathcal{M}, u \models \phi$.*

Definition 1.1.5. Given structures $\mathcal{M}_u = (S_u, R_u, L_u)$, $\mathcal{M}_v = (S_v, R_v, L_v)$ with atomic propositions AP_u, AP_v, respectively, we say $\mathcal{M}_u \equiv_{AP_v} \mathcal{M}_v$ iff

1. $AP_v \subseteq AP_u$
2. there exists a bijection $\mathbf{h} : S_u \to S_v$ such that $\mathbf{h}(u) = v$ iff $L_u(u)|_{AP_v} = L_v(v)$, where $L_u(u)|_{AP_v}$ is the set of atomic propositions in $AP_u \cap AP_v$ that label state u, and,
3. $(u, t) \in R_u$ iff $(\mathbf{h}(u), \mathbf{h}(t)) \in R_v$.

Thus, $\mathcal{M}_u$ and $\mathcal{M}_v$ have the exact same branching structure, with the only difference being in the labeling of their corresponding states. In particular, the set of labels in each state u of $\mathcal{M}_u$ is a superset of the set of labels in each corresponding state $\mathbf{h}(u)$ of $\mathcal{M}_v$. Note that since $\mathbf{h}$ is a bijection, $\mathbf{h}^{-1}$ exists. Thus, even though two states of $\mathcal{M}_u$ can be mapped into two states of $\mathcal{M}_v$ with the exact same labels, the two states of $\mathcal{M}_v$ are unique and distinguishable from each other, and can be mapped back to the two states of $\mathcal{M}_u$ via $\mathbf{h}^{-1}$.

Let $\phi(AP)$ denote a property ϕ over the set of atomic propositions AP. An obvious consequence of the above definition is the following lemma.

Lemma 1.1.4. *If $\mathcal{M}_u \equiv_{AP_v} \mathcal{M}_v$, then for any formula $\phi(AP_v)$ in CTL^*, $\mathcal{M}_u \models \phi(AP_v)$ iff $\mathcal{M}_v \models \phi(AP_v)$.*

In what follows, we fix AP to be the set $\{\mathrm{S}_{1,1}, \ldots, \mathrm{S}_{1,n_1}, \mathrm{S}_{2,1}, \ldots, \mathrm{S}_{2,n_2}\}$ of code regions of $\mathcal{P}_1$, $\mathcal{P}_2$, and AP' to be the set $\{s_{1,1}, \ldots, s_{1,n_1}, s_{2,1}, \ldots, s_{2,n_2}\}$ of all the Boolean shared variables declared in Sec. 6.3.2 to represent these code regions. Recall the states of the model $\mathcal{M}$ obtained in the first step of our algorithm in Sec. 6.3.1 are labeled with code region names and the values of shared synchronization variables $\bar{x}$. Let $\mathcal{M}'$ be a structure over AP' obtained from $\mathcal{M}$ by replacing each code region $\mathrm{S}_{k,i}$ label in every state of $\mathcal{M}$ by its corresponding counterpart $s_{k,i} \in AP'$, and suppressing the $\bar{x}$ label. We can state the following result.

Lemma 1.1.5. *For any formulas $\phi(AP)$, $\phi(AP')$ in CTL^*, $\mathcal{M} \models \phi(AP)$ iff $\mathcal{M}' \models \phi(AP')$.*

1.2 Lem. 6.3.1: Constructions and Proofs

To prove Lem. 6.3.1, we first construct refined synchronization skeletons, corresponding to $\mathcal{P}_1^c$ and $\mathcal{P}_2^c$, using the semantics of `lock(`ℓ`)`, `wait(`c,ℓ`)` and `notify(`c`)` presented in Sec. 6.1.2. We then define the global model $\mathcal{M}^c$ composed of these two synchronization skeletons, and establish the desired relation between $\mathcal{M}$ and $\mathcal{M}^c$.

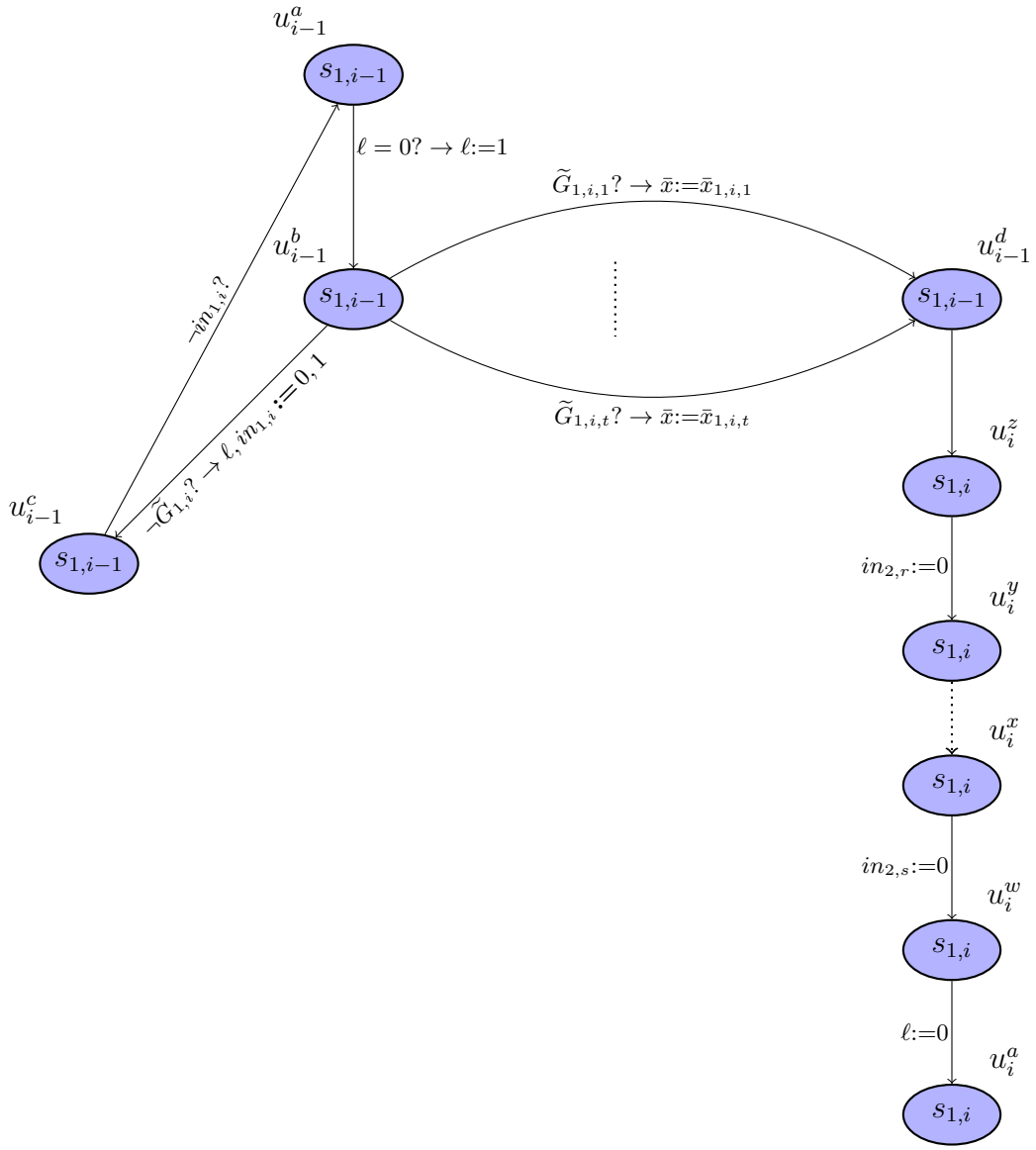


Figure 1.1: Partial refined synchronization skeleton corresponding to the implementation in Fig. 6.8a

In lieu of a formal definition, we present, in Fig. 1.1, the part of the refined synchronization skeleton that corresponds to the implementation shown in Fig. 6.8a. Each labeled transition between the states labeled with $\mathrm{S}_{1,i-1}$ and $\mathrm{S}_{1,i}$ in $\mathcal{P}_1^s$ is implemented as a series of labeled transitions between states labeled with $s_{1,i-1}$ and $s_{1,i}$ in $\mathcal{P}_1^c$. To distinguish between identically labeled states, we name the states using u_{i-1}^a, u_i^z etc. as shown in Fig. 1.1. In particular, we name any state labeled with $s_{1,i}$ as u_i^p for some p, and any state labeled with $s_{2,j}$ as v_j^p for some p (when p is not important, we leave it out and simply use u_i, v_j, respectively). Also, we denote the first state labeled with $s_{1,i}$, in which $\mathcal{P}_1$ is trying to acquire lock ℓ, as u_i^a, and the first state labeled with $s_{1,i}$, with $\mathcal{P}_1$ holding lock ℓ, as u_i^z (and similarly use v_i^a, v_i^z, for these respective states in $\mathcal{P}_2$).

Recall that our coarse-grained implementation creates a unique condition variable $cv_{1,i}$ corresponding to the overall guard $\widetilde{G}_{1,i}$ of code region $\mathrm{S}_{1,i}$, and we model the semantics of `wait(`$cv_{1,i}$`,`ℓ`)`/`notify(`$cv_{1,i}$`)` by associating a queue $q_{1,i}$ (of waiting processes) with $cv_{1,i}$. We simulate this queue $q_{1,i}$ by using a Boolean variable $in(\mathcal{P}_1, q_{1,i})$, which is set to 1 when $\mathcal{P}_1$ is added to $q_{1,i}$ and set to 0 when $\mathcal{P}_1$ is removed from $q_{1,i}$. Note that the actions in the labeled transitions of $\mathcal{P}_1^c$ do not affect the state labels as long as the actions only update the shared synchronization variables $\bar{x}$. In fact, the state label can only be changed by the parallel assignment statement $s_{1,i-1}, s_{1,i} := 0, 1$, corresponding to the transition from state u_{i-1}^d to state u_i^z. The transitions following this state label change correspond to the various notification signals

sent by $\mathcal{P}_1$ to $\mathcal{P}_2$, with $\mathcal{P}_1$ releasing the lock in the final transition into state u_i^a.

The refined synchronization skeleton for $\mathcal{P}_1^c$ (and $\mathcal{P}_2^c$) can be obtained by extending the construction in Fig. 1.1 to all synchronization regions.

We now define the global model $\mathcal{M}^c$ corresponding to $\mathcal{P}_1^c \parallel \mathcal{P}_2^c$. Let S_1^c and S_2^c denote the sets of states of the refined synchronization skeletons of $\mathcal{P}_1^c$ and $\mathcal{P}_2^c$, respectively. Let X denote the set of possible valuations of the shared synchronization variables $\bar{x}$ in $\mathcal{M}$. Let $Lk = \{0, 1, 2\}$, $IN_{1,i} = \{0, 1\}$ for $i = 1, \ldots, n_1$, and $IN_{2,j} = \{0, 1\}$ for $j = 1, \ldots, n_2$ represent the sets of possible values of the shared synchronization variables introduced in $\mathcal{P}_1^c \parallel P_2^c$. Then, $\mathcal{M}^c = (S^c, R^c, L^c)$, where $S^c = S_1^c \times S_2^c \times X \times Lk \times IN_{1,1} \times \ldots \times IN_{2,n_2}$ is the set of global states. Thus, each state w of $\mathcal{M}^c$ is a tuple of the form $(u, v, \bar{x}, \ell, in_{1,1}, \ldots, in_{2,n_2})$, with $u \in S_1^c$ and $vinS_2^c$, and $L^c(w^c)$ is an assignment of values to the members of this tuple from their respective domains. The transition relation R^c is defined such that it includes a transition from state w to state w' in $\mathcal{M}^c$ iff there exists a transition $u \xrightarrow{G \to A} u'$ in the refined synchronization skeleton for $\mathcal{P}_1^c$ such that $L^c(w)$ contains the set of labels of state u, $L^c(w')$ contains the set of labels of state u', the guard G is `true` given $L^c(w)$, and the action A results in the new valuation of the synchronization variables $\bar{x}$, as captured in $L^c(w')$, or, there exists a similar labeled transition in the refined synchronization skeleton for $\mathcal{P}_2^c$.

We present a portion of $\mathcal{M}^c$ in Fig. 1.2 that mimics the transitions shown in the refined synchronization skeleton for $\mathcal{P}_1^c$ shown in Fig. 1.1, with

the following assumptions: (1) there are no shared synchronization variables in $\mathcal{P}_1^c$, $\mathcal{P}_2^c$, (2) there is a single transition from $\mathrm{S}_{1,i-1}$ to $\mathrm{S}_{1,i}$ in $\mathcal{P}_1^s$, which is enabled on $\mathrm{S}_{2,j}$ and disabled on S_{2_k} and (3) for each code region $\mathrm{S}_{2,k}, \ldots, \mathrm{S}_{2,j+1}$ of $\mathcal{P}_2^c$, there is a single outgoing transition, which is enabled on $\mathrm{S}_{1,i-1}$. We do not show transitions of $\mathcal{M}^c$ that do not change any label, e.g., transitions corresponding to redundant notification signals sent to $\mathcal{P}_2^c$. As illustrated in Fig. 1.2, $\mathcal{M}^c$'s states can be partitioned into classes $[i,j]_h$ which can be defined as shown below (the classes in Fig. 1.2 are of the form $[i,j]$ as we assumed the absence of $\bar{x}$ variables).

Definition 1.2.1. (Class $[i,j]_h$)

1. Any state $w = (u_i, v_j, \bar{x}_h, \ldots)$ of $\mathcal{M}^c$ where $\bar{x}_h$ is the h^{th} valuation of the shared synchronization variables belongs to class $[i,j]_h$.

2. Any state $w = (u_i, v_j, \bar{x}_p, \ldots)$ of $\mathcal{M}^c$ such that $p \neq h$, and there exists a transition from a state in class $[i,j]_h$ to w belongs to class $[i,j]_h$.

3. No other state belongs to class $[i,j]_h$.

The reason we make the labels i, j in the class $[i,j]_h$ explicit, and leave the label h implicit will become clear a little later. For now, note that the classes $[i,j]_h$ induce a partition on the set of states of $\mathcal{M}^c$. We can relate this partition of $\mathcal{M}^c$ to the set of states of $\mathcal{M}$, as stated in the following lemma. We denote a state w in a class $[i,j]_h$ with no predecessors in $[i,j]_h$ as an *entry state* and with no successors in $[i,j]_h$ as an *exit state* of the class $[i,j]_h$.

Lemma 1.2.1. *[Properties of class $[i,j]_h$]*

1. *A class $[i,j]_h$ exists in $\mathcal{M}^c$ iff there exists a state $(S_{1,i}\, S_{2,j}\, \bar{x}_h)$ in $\mathcal{M}$.*

2. *A transition from a state in the class $[i,j]_h$ to a state in the class $[i+1,j]_p$ exists in $\mathcal{M}^c$ iff a transition $(S_{1,i}\, S_{2,j}\, \bar{x}_h) \rightarrow (S_{1,i+1}\, S_{2,j}\, \bar{x}_p)$ exists in $\mathcal{M}$, and the same holds true with the roles i and j reversed.*

3. *If there exists a transition $(S_{1,i}\, S_{2,j}\, \bar{x}_h) \rightarrow (S_{1,i+1}\, S_{2,j}\, \bar{x}_p)$ in $\mathcal{M}$, then there exists a path from each entry state in the class $[i,j]_h$ to an entry state in the class $[i+1,j]_p$, and the same holds true with the roles i and j reversed.*

4. *A path that starts in a state in the class $[i,j]_h$, and only involves transitions among states in the same class $[i,j]_h$, is bounded.*

Proof. We prove each part of the lemma below.

1. A state $(S_{1,i}\, S_{2,j}\, \bar{x}_h)$ exists in $\mathcal{M}$ iff there exists some transition of the form $(S_{1,i-1}\, S_{2,j}\, \bar{x}_p) \rightarrow (S_{1,i}\, S_{2,j}\, \bar{x}_h)$, or there exists some transition of the form $(S_{1,i}\, S_{2,j-1}\, \bar{x}_q) \rightarrow (S_{1,i}\, S_{2,j}\, \bar{x}_h)$ in $\mathcal{M}$. Without loss of generality, we focus our attention on the transition $(S_{1,i-1}\, S_{2,j}\, \bar{x}_p) \rightarrow (S_{1,i}\, S_{2,j}\, \bar{x}_h)$ in $\mathcal{M}$. The transition $(S_{1,i-1}\, S_{2,j}\, \bar{x}_p) \rightarrow (S_{1,i}\, S_{2,j}\, \bar{x}_h)$ exists in $\mathcal{M}$ iff there exists a transition in $\mathcal{P}_1^s$ from $S_{1,i-1}$ to $S_{1,i}$, which is enabled on $S_{2,j} \wedge \bar{x}_p$, and which assigns $\bar{x}_h$ to $\bar{x}$ (if $\bar{x}_p \neq \bar{x}_h$). This in turn is true iff there exists a transition in $\mathcal{P}_1^c$ from a state u_{i-1} to a state u_i, which is enabled

on $s_{2,j} \wedge \bar{x}_p$, and assigns $\bar{x}_h$ to $\bar{x}$. This is true iff there exists a transition $(u_{i-1}, v_j, \bar{x}_p, \ell = 1, \ldots) \rightarrow (u_i, v_j, \bar{x}_h, \ell = 1, \ldots)$ in $\mathcal{M}^c$, and hence, iff there exists a state $w = (u_i, v_j, \bar{x}_h, \ldots)$ in $\mathcal{M}^c$, in other words, iff the class $[i, j]_h$ exists in $\mathcal{M}^c$.

2. This can be proven using a similar series of arguments as above.

3. We first note from the construction of $\mathcal{M}^c$ (and the example shown in Fig. 1.2) that each path originating at an entry state in the class $[i, j]_h$ has to contain a transition to a state $(u_i, v_j, \bar{x}_h, \ell = 0, \ldots)$. Now suppose there exists a transition $(S_{1,i}\, S_{2,j}\, \bar{x}_h) \rightarrow (S_{1,i+1}\, S_{2,j}\, \bar{x}_p)$ in $\mathcal{M}$. Then, there exists a series of transitions in $\mathcal{M}^c$ (which can be inferred from the structure of the refined synchronization skeleton of $\mathcal{P}_1^c$ shown in Fig. 1.1), which start from $(u_i, v_j, \bar{x}_h, \ell = 0, \ldots)$ and lead to a state $(u_{i+1}, v_j, \bar{x}_p, \ell = 1, \ldots)$. Hence, there exists a path from each entry state in $[i, j]_h$ to a state in $[i + 1, j]_p$. The proof can be similarly applied to the case when the roles of i and j are reversed.

4. Recall that the transition relation of $\mathcal{M}$ is total. Moreover, note that there are no self-loops in $\mathcal{M}$. This follows from the interleaved model of concurrent computation in which some process with an enabled transition in its synchronization skeleton is selected to be executed next, and each transition in the synchronization skeleton of a process is from one code region to a different code region. Thus, every state in $\mathcal{M}$ has an outgoing transition to a different state in $\mathcal{M}$.

We can now prove this claim by arriving at a contradiction of the above fact. Suppose there exists a path of unbounded length within the states of class $[i,j]_h$. Since there are a finite number of states in any class, this means there exists a cycle among the states of class $[i,j]_h$. Referring to Fig. 1.2 and Fig. 1.1, this can happen iff both $\widetilde{G}_{1,i+1}$ and $\widetilde{G}_{2,j+1}$ are `false`[1]. However, if both $\widetilde{G}_{1,i+1}$ *and* $\widetilde{G}_{2,j+1}$ are `false`, both processes will be waiting on their respective condition variables, and hence, there will not be any transition out of class $[i,j]_h$. Hence, by claims (2) and (3) of this lemma, there will not be any transition out of the state $(S_{1,i}\, S_{2,j}\, \bar{x}_h)$ in $\mathcal{M}$, which contradicts the assumption that the transition relation of $\mathcal{M}$ is total. Hence, there are no cycles within the states of any class of $\mathcal{M}^c$.

□

Let $\mathcal{M}^{c'} = (S^{c'}, R^{c'}, L^{c'})$ be a structure over alphabet AP' with $\mathcal{M}^{c'} \equiv_{AP'} \mathcal{M}^c$. Thus $\mathcal{M}^{c'}$ preserves the branching structure of $\mathcal{M}^c$, with the only difference being that the labels of the states in $\mathcal{M}^{c'}$ are restricted to atoms in AP' (the labels corresponding to all the synchronization variables are suppressed in

[1]If, say, $\widetilde{G}_{2,j+1}$ is `true` and $\mathcal{P}_2$ acquires lock ℓ, then $\mathcal{P}_2$ perhaps updates the values of the shared synchronization variables to $\bar{x}_p$, and then updates the Boolean shared variables, thereby forcing a transition to a state in class $[i,j+1]_p$ within a bounded number of steps. If $\widetilde{G}_{2,j+1}$ is `true` and $\mathcal{P}_2$ does not acquire lock ℓ, i.e., $\mathcal{P}_1$ acquires lock ℓ, then either $\widetilde{G}_{2,i+1}$ is `true` and $\mathcal{P}_1$ similarly forces a transition to class $[i+1,j]_q$ within a bounded number of steps, or $\widetilde{G}_{2,i+1}$ is `false`, $\mathcal{P}_1$ yields the lock ℓ to $\mathcal{P}_2$, which forces a transition to class $[i,j+1]_p$ within a bounded number of steps.

$\mathcal{M}^{c'}$). In fact, the form of AP' is the reason why we only make the i, j labels in a class $[i,j]_h$ of $\mathcal{M}^c$ explicit. The following lemma is a direct consequence of Lem. 1.1.4.

Lemma 1.2.2. *For any formula $\phi(AP')$ in CTL^*, $\mathcal{M}^{c'} \models \phi(AP')$ iff $\mathcal{M}^c \models \phi(AP')$.*

Moreover, we can define a partition of $\mathcal{M}^{c'}$ akin to $\mathcal{M}^c$. Since $\mathcal{M}^{c'}$ is structurally the same as $\mathcal{M}^c$, Lem. 1.2.1 applies to the classes of $\mathcal{M}^{c'}$.

Lemma 1.2.3. *Any class $[i,j]_h$ of $\mathcal{M}^{c'}$ has all properties stated in Lem. 1.2.1.*

We can now state and prove the following important lemmas relating $\mathcal{M}^{c'}$ to $\mathcal{M}'$, both of which are structures over the alphabet AP'.

Lemma 1.2.4. *Let $\overline{\mathcal{M}}$ be the disjoint union of $\mathcal{M}^{c'} = (S^{c'}, R^{c'}, L^{c'})$ and $\mathcal{M}' = (S', R', L')$. There exists a stuttering simulation $B \subseteq S^{c'} \times S'$ on the states of $\overline{\mathcal{M}}$.*

Proof. Let B be a relation on the states of $\overline{\mathcal{M}}$ such that $(w^c, w) \in B$ iff w^c belongs to some class $[i,j]_h$ in $\mathcal{M}^{c'}$ and $s_{1,i}, s_{2,j}, \bar{x}_h \in L(y)$, where y is the unique state in $\mathcal{M}$ corresponding to w. Note that B is well-defined (see Lem. 1.2.3 and the definition of $\mathcal{M}'$), and that $(w^c, w) \in B$ implies $L^{c'}(w^c) = L'(w)$. Further note that B is an equivalence relation on the states of $\overline{\mathcal{M}}$, where each equivalence class $[i,j]_{h_B}$ is the union of the equivalence class $[i,j]_h$ of $\mathcal{M}^{c'}$, and exactly one state w from S' such that $L'(w) = \{s_{1,i}, s_{2,j}\}$, and $\bar{x}_h \in L(y)$ for the corresponding y. Now let $rank(a,b)$, be defined as follows:

1. $rank(a, b) = 0$ if a has a successor outside its equivalence class, and,

2. $rank(a, b) = 1 + max_c\, rank(c)$, where c is a successor of a in the same equivalence class.

As defined above, $rank$ is a well-founded function, whose maximum value is bounded by Lem. 1.2.3, and minimum value is 0 by definition.

Consider a state $w^c \in S^{c'}$ which belongs to the equivalence class $[i, j]_{h_B}$, and let $w \in S'$ be a state in the same class, i.e., $(w^c, w) \in B$. Let $w^{c'}$ be an arbitrary successor of w^c. We have the following cases:

1. $rank(w^c) = 0$: This implies that $w^{c'}$ lies outside the class $[i, j]_{h_B}$ as from the structure of $\mathcal{M}^{c'}$, the exit state w^c can have exactly one successor. In fact, either $w^{c'} \in [i + 1, j]_{p_B}$ or $w^{c'} \in [i, j + 1]_{q_B}$, for some p and q. Suppose $w^{c'} \in [i + 1, j]_{p_B}$. Then, from Lem. 1.2.3, we know that there exists a successor w' of w such that $w' \in [i+1, j]_{p_B}$. Hence, $(w^{c'}, w') \in B$. The case when $w^{c'} \in [i, j + 1]_{q_B}$ can be handled similarly.

2. $rank(w^c) \neq 0$: This implies that w^c has a successor $w^{c'} \in [i, j]_{h_B}$ such that $(w^{c'}, w) \in B$, and $rank(w^{c'}, w) \leq rank(w^c, w) - 1$, by the definition of $rank$.

Thus, we meet the requirements of Def. 1.1.4 for all successors of w^c for any $(w^c, w) \in B$. Hence, B is a well-founded simulation, and a stuttering simulation (by Lem. 1.1.2) over $\overline{\mathcal{M}}$. □

Lemma 1.2.5. *Given an* $\mathsf{ACTL}^* \setminus \mathsf{X}$ *formula* $\phi(AP')$, $\mathcal{M}' \models \phi(AP') \Rightarrow \mathcal{M}^{c'} \models \phi(AP')$.

Proof. This follows from *Lem.* 1.1.3 and *Lem.* 1.2.4. □

Proof. Lem. 6.3.1:
This follows from Lem. 1.2.2, Lem. 1.1.5 and Lem. 1.2.5, and by replacing each AP' state label in $\mathcal{M}^c$ by its corresponding counterpart in AP. □

As mentioned in Sec. 6.3.3, the translation from $\mathcal{M}$ to $\mathcal{M}^c$ preserves a larger class of properties than just $\mathsf{ACTL} \setminus \mathsf{X}$. This claim in already validated by Lem. 1.2.5, which proves preservation of $\mathsf{ACTL}^* \setminus \mathsf{X}$ properties. We present the following lemmas to further support this claim.

Lemma 1.2.6. $\mathcal{M}^{c'} \equiv_{SPE} \mathcal{M}'$.

Proof. The first part of the proof involves proving that for every path starting from an initial state w_0^c in $\mathcal{M}^{c'}$, there exists a stuttering equivalent path starting from an initial state w_0 in $\mathcal{M}'$. This follows directly from Lem. 1.2.4.

The second part of the proof involves proving the reverse direction. This follows from (1) and (3) of Lem. 1.2.1 and Lem. 1.2.3. □

Lemma 1.2.7. *Given a* $\mathsf{CTL}^* \setminus \mathsf{X}$ *formula* $\phi(AP)$ *of the form* $\mathsf{A}\psi(AP)$ *or* $\mathsf{E}\psi(AP)$, *where* $\psi(AP)$ *is an* $\mathsf{LTL} \setminus \mathsf{X}$ *formula over* AP, $\mathcal{M}^c \models \phi(AP)$ *iff* $\mathcal{M} \models \phi(AP)$.

Proof. This follows from Lem. 1.2.2, Lem. 1.1.5, Lem. 1.1.1 and Lem. 1.2.6, and again, by replacing each AP' state label in $\mathcal{M}^c$ by its corresponding counterpart in AP. □

1.3 Lem. 6.3.2: Constructions and Proofs

We only provide a brief treatment of the constructions and proofs involved in proving Lem. 6.3.2 as they are similar in spirit to those involved in proving Lem. 6.3.1. As before, to prove Lem. 6.3.2, we first construct refined synchronization skeletons, corresponding to the implementations $\mathcal{P}_1^f$ and $\mathcal{P}_2^f$. Again, in lieu of a definition, we present, in Fig. 1.3, the refined synchronization skeleton that corresponds to the implementation shown in Fig. 7.4b. The basic structure of the refined synchronization skeleton remains the same, except for the sequences of transitions corresponding to acquisition and release of multiple locks (instead of just one lock). The global model $\mathcal{M}^f$ can be defined similar to $\mathcal{M}^c$ as well, with the difference being in the larger number of states in $\mathcal{M}^f$. Let S_1^f and S_2^f denote the sets of states of the refined synchronization skeletons of $\mathcal{P}_1^f$ and $\mathcal{P}_2^f$, respectively. Let $Lk_{cv_{1,1}} = \{0,1,2\}, \ldots, Lk_{cv_{2,n_2}} = \{0,1,2\}$ represent the values of the lock variables associated with the condition variables $cv_{1,1}, \ldots, cv_{2,n_2}$, respectively. Let $Lk_{s_{1,1}} = \{0,1,2\}, \ldots, Lk_{s_{2,n_2}} = \{0,1,2\}$ represent the values of the lock variables associated with the Boolean shared variables $s_{1,1}, \ldots, s_{2,n_2}$, respectively. Thus, $\mathcal{M}^f = (S^f, R^f, L^f)$, where $S^f = S_1^f \times S_2^f \times X \times Lk_{cv_{1,1}} \times \ldots \times Lk_{cv_{2,n_2}} \times Lk_{s_{1,1}} \times \ldots \times Lk_{s_{2,n_2}} \times IN_{1,1} \times \ldots \times IN_{2,n_2}$, L^f is an assignment of values from appropriate domains to each member of a

state tuple, and R^f is defined similar to R^c.

Having defined the refined synchronization skeletons corresponding to $\mathcal{P}_1^f$ and $\mathcal{P}_2^f$, and $\mathcal{M}^f$, we can now extend all the definitions and lemmas leading to the proof of Lem. 6.3.1, to corresponding definitions and lemmas leading to the proof of Lem. 6.3.2.

The states of $\mathcal{M}^f$ can be partitioned into classes $[i, j]_h$ defined according to Def. 1.2.1, which in turn can be related to the states of $\mathcal{M}$ as stated in Lem. 1.2.1. The proof of Lem. 1.2.1 for the classes of $\mathcal{M}^f$ is similar in spirit to that presented already. Intuitively, this is due to the similarity in the structures of the refined synchronization skeletons for $\mathcal{P}_1^c$, $\mathcal{P}_2^c$ and $\mathcal{P}_1^f$, $\mathcal{P}_2^f$. One only needs to account for a longer sequence of transitions corresponding to acquisition and release of locks in the proof.

The rest of the lemmas: Lem. 1.2.2, Lem. 1.2.3, Lem. 1.2.4 and Lem. 1.2.5 extend in a straight-forward manner, leading to the proof of Lem. 6.3.2.

Note that Lem. 1.2.6 and Lem. 1.2.7 also hold for the fine-grained case.

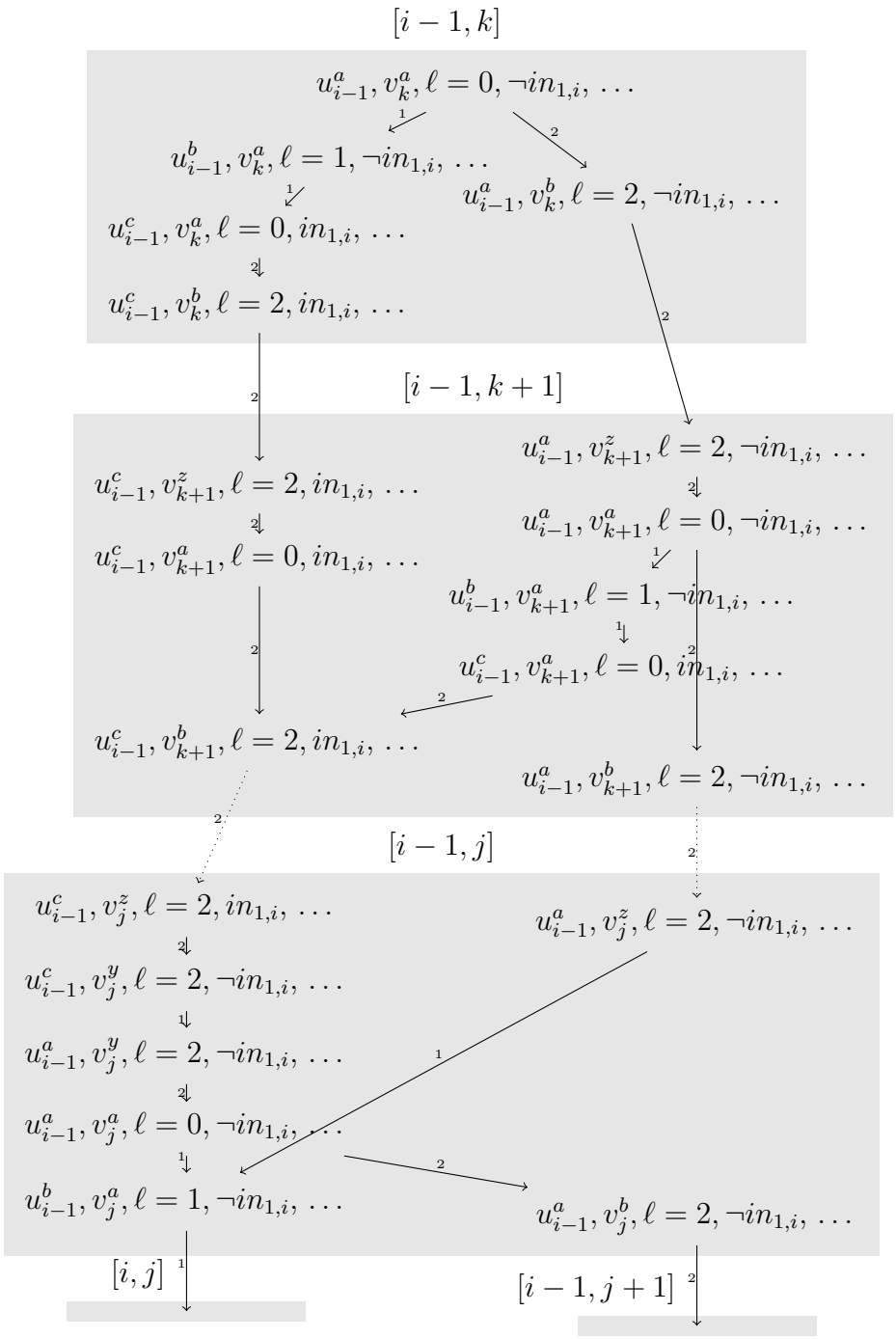


Figure 1.2: A partial representation of $\mathcal{M}^c$

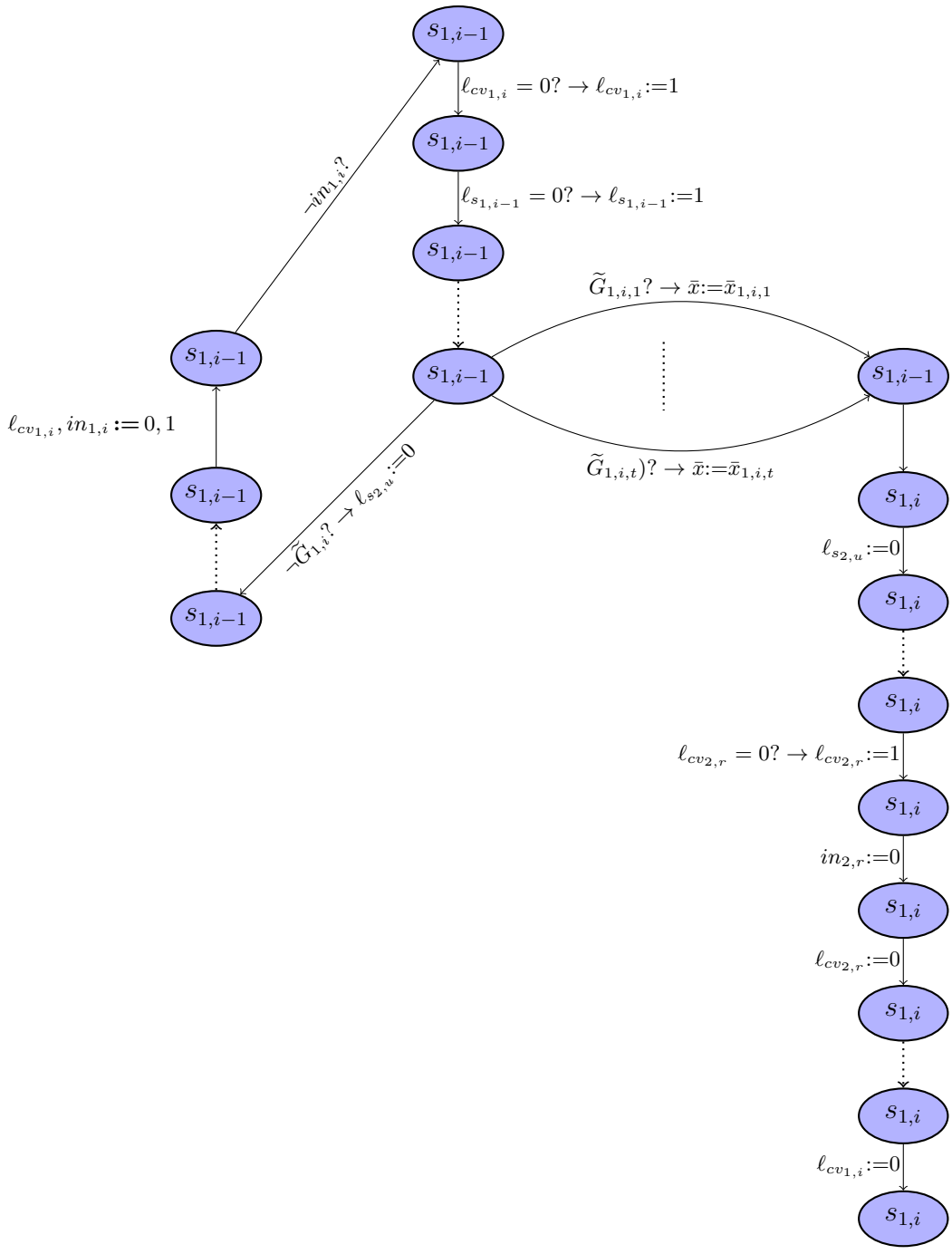


Figure 1.3: Refined synchronization skeleton corresponding to implementation in Fig. 6.8b

Bibliography

[1] Rajeev Alur, Alessandro D'Innocenzo, Karl Henrik Johansson, George J. Pappas, and Gera Weiss. Compositional Modeling and Analysis of Multi-Hop Control Networks. *IEEE Transactions on Automatic Control*, 56(10):2345–2357, 2011.

[2] A. Arcuri. On the Automation of Fixing Software Bugs. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 1003–1006. ACM, 2008.

[3] P. C. Attie. Synthesis of Large Concurrent Programs via Pairwise Composition. In *Proceedings of International Conference on Concurrency Theory (CONCUR)*, pages 130–145. ACM, 1999.

[4] P. C. Attie and E. A Emerson. Synthesis of Concurrent Systems with Many Similar Sequential Processes. In *Proceedings of Principles of Programming Languages (POPL)*, pages 191–201. ACM, 1989.

[5] P. C. Attie and E. A Emerson. Synthesis of Concurrent Systems for an Atomic Read/Atomic Write Model of Computation. In *Proceedings of Principles of Distributed Computing (PODC)*, pages 111–120. ACM, 1996.

[6] P. C. Attie and J. Saklawi. Model and Program Repair via SAT Solving. *CoRR*, abs/0710.3332, 2007.

[7] A. Auer, J. Dingel, and K. Rudie. Concurrency Control Generation for Dynamic Threads using Discrete-Event Systems. In *Proceedings of the Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 927–934. IEEE, 2009.

[8] M. Autili, P. Inverardi, A. Navarra, and Massimo Tivoli. SYNTHESIS: A Tool for Automatically Assembling Correct and Distributed Component-Based Systems. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 784–787. ACM, 2007.

[9] T. Ball. Formalizing Counterexample-driven Refinement with Weakest Preconditions. In *Proceedings of Engineering Theories of Software Intensive Systems*, pages 121–139. Springer-Verlag, 2005.

[10] T. Ball, E. Bounimova, R. Kumar, and V. Levin. SLAM2: Static Driver Verification with under 4% False Alarms. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD)*, pages 35–42, 2010.

[11] T. Ball, M. Naik, and S. K. Rajamani. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *Proceedings of Principles of Programming Languages (POPL)*, pages 97–105. ACM, 2003.

[12] T. Ball and S. K. Rajamani. Boolean Programs: A Model and Process for Software Analysis. Technical Report MSR-TR-2000-14, Microsoft

Research, 2000.

[13] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Proceedings of the International Workshop on Model Checking of Software (SPIN)*, pages 103–122. Springer-Verlag, 2001.

[14] D. Beyer, T. A. Henzinger, and V. Singh. Algorithms for Interface Synthesis. In *Proceedings of Computer Aided Verification (CAV)*, pages 4–19. Springer-Verlag, 2007.

[15] R. Bloem, K. Chatterjee, T. A. Henzinger, and B. Jobstmann. Better Quality in Synthesis through Quantitative Objectives. In *Proceedings of Computer Aided Verification (CAV)*, pages 140–156. Springer, 2009.

[16] R. Bloem, K. Greimel, T. Henzinger, and B. Jobstmann. Synthesizing Robust Systems. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD)*, pages 85–92, 2009.

[17] E. Bonta, M. Bernardo, J. Magee, and J. Kramer. Synthesizing Concurrency Control Components from Process Algebraic Specifications. In *Proceedings of the International Conference on Coordination Models and Languages*, pages 28–43, 2006.

[18] F. Boussinot and R. De Simone. The ESTEREL Language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[19] Robert K. Bradley and Ian Holmes. Transducers: An Emerging Probabilistic Framework for Modeling Indels on Trees. *Bioinformatics*, 23(23):3258–3262, 2007.

[20] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. Enhancing Model Checking in Verification by AI Techniques. *Artificial Intelligence*, 112(1-2):57–104, 1999.

[21] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic Debugging. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 121–130. ACM, 2011.

[22] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity Analysis of Programs. In *Proceedings of Principles of Programming Languages (POPL)*, pages 57–70, 2010.

[23] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity and Robustness of Programs. *Communications of the ACM*, 2012.

[24] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving Programs Robust. In *Proceedings of Foundations of Software Engineering*, pages 102–112, 2011.

[25] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring Locks for Atomic Sections. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 304–315. ACM, 2008.

[26] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based Predicate Abstraction for ANSI-C. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 570–574. Springer Verlag, 2005.

[27] E. M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Proceedings of Logics of Programs*, volume 131, pages 52–71. Springer Berlin Heidelberg, 1982.

[28] M. A. Colón. Schema-Guided Synthesis of Imperative Programs by Constraint Solving. In *Proceedings of Logic Based Program Synthesis and Transformation (LOPSTR)*, volume 3573, pages 166–181. Springer Berlin Heidelberg, 2005.

[29] L. de Alfaro and T. A. Henzinger. Interface automata. *SIGSOFT Software Engineering Notes*, 26(5):109–120, 2001.

[30] V. Debroy and W. E. Wong. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *Proceedings of Software Testing, Verification and Validation (ICST)*, pages 65–74, 2010.

[31] S. Demri and D. D'Souza. An Automata-theoretic Spproach to Constraint LTL. *Information and Computation*, 205(3):380–415, 2007.

[32] X. Deng, M. B. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based Specification, Synthesis, and Verification of Synchronization in Concur-

rent Programs. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 442–452. ACM, 2001.

[33] L. A. Dennis. Program Slicing and Middle-Out Reasoning for Error Location and Repair. In *Proceedings of Disproving: Non-Theorems, Non-Validity and Non-Provability*, 2006.

[34] L. A. Dennis, R. Monroy, and P. Nogueira. Proof-directed Debugging and Repair. In *Proceedings of the Symposium on Trends in Functional Programming*, pages 131–140, 2006.

[35] J. V. Deshmukh, E. A. Emerson, and R. Samanta. Economical Transformations for Structured Data. Technical Report TR-10-28, The University of Texas at Austin, Department of Computer Sciences, 2010.

[36] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[37] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag New York, Inc., 1990.

[38] I. Dillig, T. Dillig, and A. Aiken. Automated Error Diagnosis using Abductive Inference. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 181–192. ACM, 2012.

[39] H. Do, S. G. Elbaum, and G. Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal*, 2005.

[40] L. Doyen, T. A. Henzinger, A. Legay, and D. Ničković. Robustness of Sequential Circuits. In *Proceedings of Application of Concurrency to System Design (ACSD)*, pages 77–84, 2010.

[41] C. Dragert, J. Dingel, and K. Rudie. Generation of Concurrency Control Code using Discrete-Event Systems Theory. In *Proceedings of International Symposium on Foundations of Software Engineering (FSE)*, pages 146–147. ACM, 2008.

[42] A. Ebnenasir, S. S. Kulkarni, and B. Bonakdarpour. Revising UNITY Programs: Possibilities and Limitations. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 275–290, 2005.

[43] John C. Eidson, Edward A. Lee, Slobodan Matic, Sanjit A. Seshia, and Jia Zou. Distributed Real-Time Software for Cyber-Physical Systems. *Proceedings of the IEEE (special issue on CPS)*, 100(1):45–59, 2012.

[44] Samuel Eilenberg. *Automata, Languages, and Machines*, volume A. Academic Press New York, 1974.

[45] E. A Emerson, S. Jha, and D. Peled. Combining Partial Order and Symmetry Reductions. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 19–34, 1997.

[46] E. A. Emerson, T. Sadler, and J. Srinivasan. Efficient Temporal Reasoning. In *Proceedings of Principles of Programming Languages (POPL)*, pages 166–178. ACM, 1989.

[47] E. Allen Emerson and E. M. Clarke. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[48] E. Allen Emerson and R. Samanta. An Algorithmic Framework for Synthesis of Concurrent Programs. In *Proceedings of Automated Technqiues for Verification and Analysis (ATVA)*, pages 522–530, 2011.

[49] M. Emmi, J. S. Fishcher, R. Jhala, and R. Majumdar. Lock Allocation. In *Proceedings of Principles of Programming Languages (POPL)*, pages 291–296, 2007.

[50] E. Ermis, M. Schäf, and T. Wies. Error Invariants. In *Proceedings of Formal Methods (FM)*, pages 187–201. Springer Berlin Heidelberg, 2012.

[51] H. R. Nielson F. Nielson and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[52] R. W. Floyd. Assigning Meanings to Programs. In *Proceedings of Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[53] C. Frougny and J. Sakarovitch. Rational Relations with Bounded Delay. In *Proceedings of Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 50–63, 1991.

[54] R. Gascon. An Automata-based Approach for CTL* With Constraints. *Electronic Notes in Theoretical Computer Science*, 239:193–211, 2009.

[55] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE Press, 2012.

[56] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *Proceedings of Computer Aided Verification (CAV)*, pages 72–83. Springer Verlag, 1997.

[57] A. Griesmayer, R. Bloem, and B. Cook. Repair of Boolean Programs with an Application to C. In *Proceedings of Computer Aided Verification (CAV)*, pages 358–371, 2006.

[58] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error Explanation with Distance Metrics. *Int. J. Softw. Tools Technol. Transf.*, 8(3):229–247, 2006.

[59] S. Gulwani. Dimensions in Program Synthesis. In *Proceedings of Principles and Practice of Declarative Programming (PPDP)*, pages 13–24. ACM, 2010.

[60] E. Gurari and O. Ibarra. A Note on Finite-valued and Finitely Ambiguous Transducers. *Mathematical Systems Theory*, 16(1):61–66, 1983.

[61] Eitan M. Gurari and Oscar H. Ibarra. The Complexity of Decision Problems for Finite-Turn Multicounter Machines. In *Proceedings of the International Colloquium on Automata Languages and Programming (ICALP)*, pages 495–505, 1981.

[62] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.

[63] P. B. Hansen. Edison – A Multiprocessor Language. *Software – Practice and Experience*, 11(4):325–361, 1981.

[64] D. Harel and A. Pnueli. On the Development of Reactive Systems. In *Proceedings of Logics and Models of Concurrent Systems*, pages 477–498, 1985.

[65] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In *Proceedings of the International Workshop on Model Checking of Software (SPIN)*, pages 235–239, 2003.

[66] C. A. R. Hoare. Towards a Theory of Parallel Programming. In *Proceedings of Operating Systems Techniques*, pages 61–71, 1971.

[67] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge Univeristy Press, 2004.

[68] Oscar H. Ibarra. Reversal-Bounded Multicounter Machines and Their Decision Problems. *Journal of the ACM*, 25(1):116–133, 1978.

[69] Oscar H. Ibarra, J. Su, Z. Dang, T. Bultan, and R. A. Kemmerer. Counter Machines: Decidable Properties and Applications to Verification Problems. In *Proceedings of Mathematical Foundations of Computer Science (MFCS)*, pages 426–435, 2000.

[70] J. Gray. Why Do Computers Stop And What Can Be Done About It? In *Proceedings of Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[71] J. Silva. A Survey on Algorithmic Debugging Strategies. *Advances in Engineering Software*, 42(11):976–991, 2011.

[72] M. U. Janjua and A. Mycroft. Automatic Correction to Safety Violations in Programs. In *Proceedings of Thread Verification*, 2006.

[73] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated Atomicity-violation Fixing. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 389–400. ACM, 2011.

[74] H. Jin, K. Ravi, and F. Somenzi. Fate and Free Will in Error Traces. *International Journal on Software Tools for Technology Transfer*, 6(2):102–116, 2004.

[75] B. Jobstmann, A. Griesmayer, and R. Bloem. Program Repair as a Game. In *Proceedings of Computer Aided Verification (CAV)*, pages 226–238. Springer-Verlag, 2005.

[76] B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem. Finding and Fixing Faults. *Journal of Computer and System Sciences (JCSS)*, 78(2):441–460, 2012.

[77] M. Jose and R. Majumdar. Cause Clue Clauses: Error Localization using Maximum Satisfiability. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 437–446. ACM, 2011.

[78] K. Zhou and J. C. Doyle and K. Glover. *Robust and Optimal Control*. Prentice Hall, 1996.

[79] V. Kahlon. Automatic Lock Insertion in Concurrent Programs. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, pages 16–23, 2012.

[80] Gilles Kahn. The Semantics of Simple Language for Parallel Programming. In *Proceedings of IFIP Congress*, pages 471–475, 1974.

[81] R. Könighofer and R. Bloem. Automated Error Localization and Correction for Imperative Programs. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD)*, pages 91–100, 2011.

[82] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete Functional Synthesis. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 316–329. ACM, 2010.

[83] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Functional Synthesis for Linear Arithmetic and Sets. In *Proceedings of Software Tools for Technology Transfer (STTT)*, pages 455–474, 2012.

[84] M. Kuperstein, M. T. Vechev, and E. Yahav. Automatic Inference of Memory Fences. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, pages 108–123, 2010.

[85] O. Kupferman. Recent Challenges and Ideas in Temporal Synthesis. In *Proceedings of Current Trends in Theory and Practice of Computer Science (SOFSEM)*, pages 88–98. Springer Berlin Heidelberg, 2012.

[86] F. Liu, N. Nedev, N. Prisadnikov, M. T. Vechev, and E. Yahav. Dynamic Synthesis for Relaxed Memory Models. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 429–440, 2012.

[87] F. Logozzo and T. Ball. Modular and Verified Automatic Program Repair. In *Proceedings of Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 133–146. ACM, 2012.

[88] R. Majumdar and I. Saha. Symbolic Robustness Analysis. In *IEEE Real-Time Systems Symposium*, pages 355–363, 2009.

[89] Rupak Majumdar, Elaine Render, and Paulo Tabuada. A Theory of Robust Software Synthesis. *CoRR*, abs/1108.3540, 2011.

[90] Z. Manna. *Introduction to Mathematical Theory of Computation.* McGraw-Hill, Inc., 1974.

[91] 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.

[92] P. Manolios. Brief Announcement: Branching Time Refinement. In *Proceedings of Principles of Distributed Computing (PODC)*, page 334. ACM, 2003.

[93] M. Mohri. Finite-state Transducers in Language and Speech Processing. *Computational Linguistics*, 23(2):269–311, 1997.

[94] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340. Springer-Verlag, 2008.

[95] K. Namjoshi. A Simple Characterization of Stuttering Bisimulation. In *Proceedings of Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 284–296, 1997.

[96] National Institute of Standards and Technology. The Economic Impacts of Inadequate Infrastructure for Software Testing, 2002.

[97] NEC. NECLA Static Analysis Benchmarks. `http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php#NECLA_Static_Analysis_Benchmarks`.

[98] Miroslav Pajic, Shreyas Sundaram, George J. Pappas, and Rahul Mangharam. The Wireless Control Network : A New Approach for Control

Over Networks. *IEEE Transactions on Automatic Control*, 56(10):2305–2318, 2011.

[99] George J. Pappas. Wireless Control Networks: Modeling, Synthesis, Robustness, Security. In *Proceedings of Hybrid Systems: Computation and Control (HSCC)*, pages 1–2, 2011.

[100] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of Foundations of Computer Science (FOCS)*, pages 46–77, 1977.

[101] A. Pnueli. Linear and Branching Structures in the Semantics and Logics of Reactive Systems. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, pages 15–32, 1985.

[102] A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *Proceedings of Principles of Programming Languages (POPL)*, pages 179–190. ACM, 1989.

[103] A. Pnueli and R. Rosner. Distributed Reactive Systems are Hard to Synthesize. In *Proceedings of Foundations of Computer Science (FOCS)*, pages 746–757, 1990.

[104] R. M. Prasad and A. Biere and A. Gupta. A Survey of Recent Advances in SAT-based Formal Verification. *International Journal on Software Tools for Technology Transfer*, 7(2):156–173, 2005.

[105] M. Renieris and S. P. Reiss. Fault Localization With Nearest Neighbor Queries. In *Proceedings of Automated Software Engineering (ASE)*, pages 30–39, 2003.

[106] R. Samanta. Towards Algorithmic Synthesis of Synchronization for Shared-Memory Concurrent Programs. In *Proceedings of the Workshop on Synthesis (SYNT)*, volume 84 of *EPTCS*, pages 17–32, 2012.

[107] R. Samanta, J. V. Deshmukh, and S. Chaudhuri. Robustness Analysis of Networked Systems. In *Proceedings of Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 229–247, 2013.

[108] R. Samanta, J. V. Deshmukh, and S. Chaudhuri. Robustness Analysis of String Transducers. In *Proceedings of Automated Technqiues for Verification and Analysis (ATVA)*, pages 427–441, 2013.

[109] R. Samanta, J. V. Deshmukh, and E. A. Emerson. Automatic Generation of Local Repairs for Boolean Programs. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10, 2008.

[110] R. Samanta, O. Olivo, and E. A. Emerson. Cost-Aware Automatic Program Repair. *CoRR*, abs/1307.7281, 2013.

[111] E. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1982.

[112] R. Singh, S. Gulwani, and A. Solar-Lezama. Automatic Feedback Generation for Introductory Programming Assignments. In *Proceedings of*

Programming Language Design and Implementation (PLDI), pages 15–26, 2013.

[113] R. Singh and A. Solar-Lezma. Synthesizing Data-Structure Manipulations from Storyboards. In *Proceedings of Foundations of Software Engineering (FSE)*, pages 289–299, 2011.

[114] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching Concurrent Data Structures. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 136–148. ACM, 2008.

[115] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu. Programming by Sketching for Bit-streaming Programs. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 281–294. ACM, 2005.

[116] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial Sketching for Finite Programs. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415. ACM, 2006.

[117] S. Srivastava, S. Gulwani, and J. S. Foster. From Program Verification to Program Synthesis. In *Proceedings of Principles of Programming Languages (POPL)*, pages 313–326. ACM, 2010.

[118] T. Ball. The Concept of Dynamic Analysis. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIG-*

SOFT Foundations of Software Engineering (ESEC/FSE), pages 216–234. Springer-Verlag, 1999.

[119] P. Tabuada, A. Balkan, S. Y. Caliskan, Y. Shoukry, and R. Majumdar. Input Output Stability for Discrete Systems. In *Proceedings of International Conference on Embedded Software (EMSOFT)*, 2012.

[120] W. Thomas. Facets of Synthesis: Revisiting Church's Problem. In *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS)*, pages 1–14. Springer-Verlag, 2009.

[121] V. D'Silva and D. Kroening and G. Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions of Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[122] M. Y. Vardi and P. Wolper. Reasoning about Infinite Computations. *Information and Computation*, 115(1):1–37, 1994.

[123] P. Černý, T. A. Henzinger, and A. Radhakrishna. Simulation Distances. In *Proceedings of International Conference on Concurrency Theory CONCUR*, pages 253–268, 2010.

[124] P. Černý, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Efficient Synthesis for Concurrency by Semantics-preserving Transformations. In *Proceedings of Computer Aided Verification (CAV)*, pages 951–967. Springer-Verlag, 2013.

[125] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjørner. Symbolic Finite State Transducers: Algorithms and Applications. In *Proceedings of Principles of Programming Languages (POPL)*, pages 137–150, 2012.

[126] M. T. Vechev, E. Yahav, and G. Yorsh. Inferring Synchronization under Limited Observability. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 139–154, 2009.

[127] M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-Guided Synthesis of Synchronization. In *Proceedings of Principles of Programming Languages (POPL)*, pages 327–388, 2010.

[128] C. von Essen and B. Jobstmann. Program Repair without Regret. In *Proceedings of Computer Aided Verification (CAV)*, pages 896–911. Springer Berlin Heidelberg, 2013.

[129] Y. Wang, S. Lafortune, T. Kelley, M. Kudlur, and S. Mahlkeh. The Theory of Deadlock Avoidance via Discrete Control. In *Proceedings of Principles of Programming Languages (POPL)*, pages 252–263, 2009.

[130] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated Fixing of Programs with Contracts. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, pages 61–72. ACM, 2010.

[131] J. Whaley. JavaBDD: An Efficient BDD Library for Java. `http://javabdd.sourceforge.net/`.

[132] T. Yavuz-Kahveci and T. Bultan. Specification, Verification, and Synthesis of Concurrency Control Components. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, pages 169–179. ACM, 2002.

[133] R. N. Zaeem, D. Gopinath, S. Khurshid, and K. S. McKinley. History-Aware Data Structure Repair using SAT. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 2–17. Springer-Verlag, 2012.

[134] A. Zeller and R. Hilebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.

[135] Y. Zhang and Y. Ding. CTL Model Update for System Modifications. *Journal of Artificial Intelligence Research*, 31:113–155, 2008.